Developing Database Applications using the PDSA System

This chapter will discuss our database design philosophy and why we approach our database applications with a specified design pattern. It is recommended that you follow this approach when using the PDSA .NET Productivity Framework. Our tools are designed to be used with this approach and you will find your development will be much simpler if you follow the guidelines set forth in this chapter.

## Database Design

When designing your database, as with most things in programming, it is best to be consistent. This consistency applies to how you name your tables, stored procedures, triggers, column names, indexes and any database objects you create. See our PDSA Database Standards document in the appendix for more information about using these standards.

This document describes how PDSA creates all of our database applications for maximum efficiency, performance, and ease of programming. The primary focus of our system is on consistency and design patterns.

## Primary Keys

How you choose to create primary keys in your tables will affect many things about your database applications. It can affect performance, size of the database, and maintenance. Choose the primary key for your tables wisely and you will find yourself having an easier time developing applications. Let's discuss many of the different methods you can use to generate primary keys and then look at the advantages and disadvantages of each.

### Using Data Column(s)

You could choose to use one of your regular data columns as your primary key. For example, if you have a Customers table, you might use a customer number as the primary key. If this customer number is an integer number that is unique that is fine. If this customer number is a string, then you face some performance penalties when performing a JOIN to another table that has a foreign key reference to this customer number. The Integer data type is the best type to use for primary keys as it is the fastest for database systems to join on.

**TIP**: Use an Integer data type for all primary keys. It is the fastest when performing JOINs across tables.

Concatenated keys for primary keys should be avoided at all costs. There are a few problems with using more than one column as a primary key.

If you are going to have another table that will have a foreign key reference, all those fields must be duplicated from one table to another. This takes up extra space in the database and on the hard disk.

Performance will suffer when doing JOINs. It will simply take a lot longer for the JOIN to occur.

If one of the primary key values changes you will need to perform a cascading update to all related tables. This is a very expensive operation.

**TIP**: Avoid Concatenated keys for Primary Keys

### Using a Surrogate Key

The best method that we have found over the years for a primary key is to always use an Integer data type that is simply an incrementing number. While your data column(s) is still considered your "primary key" to the database, this unique id field is the primary key. This is sometimes called a surrogate key. Using a surrogate key like this solves many of the problems discussed in the previous section. First, an Integer data type is the fastest way to join tables together. Second, it is only 8 bytes of data to carry from one table to a related table. This means you are keeping your database size to a minimum. Third, you will never need to change this value so you avoid the whole cascade update problem.

Of course there is a downside to using this approach. When you are looking at your data, especially across foreign key tables, it is not as easy to see the related data. For example, if you had a table that had orders for your customer, you would need to do a join with the customer table to get the real customer number for the orders. While this is not difficult to do, it does need someone who is a little more database savvy to perform this operation.

When you choose to use a surrogate key you now have to figure out how to generate this key. Most database systems today have a method to auto-generate a primary key like this. In SQL Server it is called an **IDENTITY**. In Oracle, it is called a **Sequence Number**.

### The pdsaTableIDs Table

Of course you could also roll your own auto-increment routine as well. We have built into our data classes and our framework the ability to generate a new Integer primary key using a similar construct to a SQL Server IDENTITY, an Oracle Sequence Number and an Access AutoNumber. To do this we use a table called pdsaTableIDs and a method in our PDSADataLayer class called GetNewPrimaryKey.

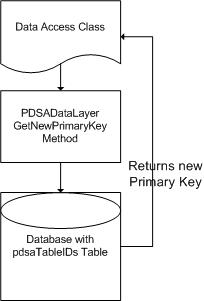


Figure 1. The pdsaTableIDs table keeps track of the next IDs to generate

There are some good reasons why you might choose to use this auto numbering method instead of using any of the built in methods.

1. This model is generic across any database platform. This means the same code works regardless of whether we use SQL Server, Oracle, Sybase or Access.
2. Our method makes it very easy to reset the next ID to generate. We simply wipe out our pdsaTableIDs table and our data layer figures out the next ID to generate.
3. This method will work with XML files, NoSQL databases, and just about any other storage mechanism.
4. With IDENTITY or Sequence Number, you have to submit the SQL first, before you get the number. This makes transactions harder to do. For example, if you want to do a transaction across a parent-child relationship, you have to write the transaction code yourself around the call to the parent and the child INSERTs. You must get the parent IDENTITY/Seq. Number first, then pass it to the childs. All of this coding has to be done within a transaction. Using our method, we can retrieve all of the primary keys first, give them to the child objects, then pass all the objects to a generic method in our PDSATransaction class to submit all the SQL at one time with all primary and foreign keys filled in. This cuts down the amount of code you have to write.

### PDSA Recommendation

Always use a surrogate key and either use SQL IDENTITY, or use our pdsaTableIDs to generate those keys. All of our tools are setup to use both models.

There are some tables on which you won't want to use an auto-number field. Tables such as States or Countries where you have a State Code or a Country Code and you just want to put the state code into the Customers table. That is fine. But for all your other tables, you should always do this.

## Clustered Index

Be sure that each table has a clustered index on it. It probably should not be the primary key. It is better to use a field that will be searched upon most often. Having a clustered index is important to the SQL optimizer. If you do not have a clustered index, then a table scan will most often be used for searching, even if you have an index that would cover the search.

## Standard Fields

Over the years we have worked with many clients who needed to track what changes were made to what records. To accomplish this, it is necessary for you to add some fields to each table so you can track who last updated the record and the date and time on which they updated it. In addition, you may wish to track who inserted the record, and the date on which it was inserted.

Once again, our tools are setup so that each table is consistent within the database. So if you add some of these standard fields to your tables, be sure to do it on ALL tables. And we mean ALL TABLES! Even if you have a table with just state codes or country codes, be sure to add these fields to those too.

### PDSA Recommendation

Below are the fields that we recommend that you add to each table.

| Field Name | Data Type | Description |
| --- | --- | --- |
| InsertName | varchar(50) | Who inserted the record |
| InsertDate | Datetime | Date the record was inserted |
| UpdateName | varchar(50) | User who last updated the record. |
| UpdateDate | datetime | Date the record was last updated |
| ConcurrencyNumber | smallint | A concurrency tracking number. This is explained later in this document. |

Table 1. Table of standard fields you should put into each table.

NOTE: After adding the standard fields, you should run UPDATE <tableName> SET ConcurrencyNumber = 1 to ensure that this value is filled in.

TIP: If you are using SQL Server there is a SQL file that will add these fields to all the tables within a database. Just look in the \PDSAFramework-<verno>\Database\SQLServerScripts\Misc for the AddStandardFieldsToTables.sql file. Load this into SQL Query Analyzer and run it to get the SQL to run in the results window. Copy the results window to the clipboard and copy back into a new Query windows and run the script.

## Handling Concurrency

If you have more than one user for an application, you will need to think about how to handle the situation where two users are trying to update the same record at the same time. There are many different techniques you can use to handle this situation. The method that PDSA employs is to include a column in each table (called ConcurrencyNumber) that holds a small integer value. When a record is read by each user, this value is read and stored into memory for that user. When User A and User B both read a record they will have the same number for this column, let's say it is the value one (1). When User A updates the record first, part of the UPDATE statement will increment this value by one. The WHERE clause for updating this record not only includes the primary key value, but also the value read in for the concurrency id. See the example below.

UPDATE States

SET sStateName = 'California'

ConcurrencyNumber = ConcurrencyNumber + 1

WHERE sStateCode = 'CA'

AND ConcurrencyNumber = 1

Now, when User B attempts to update this record, with some other value, that user's update statement will not affect any rows because the ConcurrencyNumber is now a 2 not a 1 that User B is looking for. You can check for the number of rows affected and if it is zero, then you can inform User B that the data has changed since the time that they read that record.

## Dynamic SQL or Stored Procedures

There is always an ongoing debate about whether to use dynamic SQL or stored procedures for your data access. There are advantages and disadvantages to both. Let's take a look at what these are.

### Advantages of Dynamic SQL

When looking at your source code you can see exactly what is going to be submitted to the database without having to look up a stored procedure name in the database.

It is much easier to reuse the same SELECT statement over and over again with any combination of WHERE clause and/or ORDER BY. With stored procedures you need a unique stored proc for each combination, or you need to build the SQL dynamically in the stored proc.

It is easier to perform query-by-example searches since you can build the WHERE clause dynamically from the user input right in your application.

There are two places you can keep business logic surrounding the SQL for a database entity. By keeping your SQL outside of stored procedures you keep your business logic in business objects adhering strictly to n-tier design principles.

The advantage of keeping your business rules in business objects is that you have access to a higher–level language for applying the rules.

### Disadvantages of Dynamic SQL

Performance is not as good as stored procedures

You must grant rights to base tables to a specific user id. If anyone discovers this id and password then they can get into your data.

### Advantages of Stored Procedures

They are very fast to execute compared to Dynamic SQL since they are precompiled.

All the logic is stored in one place making changes easier.

A DBA can tune stored procedures in one place, and all applications that use those stored procs can then get these performance benefits without recompiling.

You can just grant rights to execute stored procedures to a specific user id. This id does not need rights to the base tables. This is much more secure.

### Disadvantages of Stored Procedures

When programming you have to remember the names of a lot of stored procedures.

You have to manage all these stored procedures and ensure they are checked into your version control system.

### PDSA Recommendation

PDSA uses both techniques depending on what the application needs. We find a mixture of both is typically very acceptable for most normal business applications. Unless your application is going to service thousands of users and you need the utmost in performance and security, you will find Dynamic SQL works just fine without all the management stored procedures requires.

## Triggers

It is highly recommended that you avoid the use of triggers. There are not too many instances where you would need to use them anymore.

## N-Tier

It is highly recommended that you use N-Tier techniques when developing your applications. This means that you separate your applications into separate services. Each service performs a specific function. For example, you will have one service that does your UI logic. You will have another service that performs business process logic. You have another service that may enforce business rules for a table or set of tables. You have another service that manages the SQL/stored procedures calls for each table or set of tables. You have another service that handles the most efficient method of submitting SQL to the back end database. You finally have your database service. This is a lot of services, but this type of design is very efficient and leads to great reusability and maintainability.

## Reporting

When it is time to develop reports for your applications, there are some best practices that you will find will help you in this effort. For example, you should always create stored procedures and/or views to normalize the data that will be fed to the report. When designing a report ensure you do not connect the report directly to the database. Instead use an XSD file to define the complete schema of all the columns you will need for the report. You then create a DataSet and feed that to the report at runtime. If you follow this method, it will allow you to change from one reporting tool to another will little effort.

Always perform as much data gathering on the server as possible. Reporting tools are good for formatting data, but they are typically horrible at manipulating data. The most efficient reports are those that let the database server do the data manipulation, and just feed the data to the reporting tool for formatting.

Summary

In this chapter you learned some best practices for developing database applications. PDSA follows these practices in every application we develop. It is highly recommended that you follow these practices as well to get the most out of the PDSA .NET Productivity Framework.