ASSESSING DATABASES IN .NET
Comparing Approaches

Daniela da Cruz and Pedro Rangel Henriques
Department of Computer Science, University of Minho, Braga, Portugal

Keywords: Databases, LINQ, LINQDataSource, SQLDataSource, ObjectDataSource, Performance.

Abstract: Language-Integrated Query (LINQ) appeared recently as the new language of the .NET framework — *is the new kid of the town*.
This query-language, an extension to C# and Visual Basic, allows the query expressions to benefit from the features previously available only to imperative code — the rich metadata, IntelliSense, compile-time syntax checking, and static typing.
In this paper, we intend to compare the methods provided by .NET to query databases (LINQ, SQL and Object). This comparison will be done in terms of performance and in terms of the approach used. To guide this comparison, a running-example will be used.

1 INTRODUCTION

When writing programs to implement database-centered Information Systems (IS), *operations to access databases* are frequent and important, but inefficient. The importance derives from the fact that almost all the work in a IS is based on transactions of data to or from the database; the inefficiency is due to the time inherent to read/write operations from/to the disk.

This is even more critical in the context of Web Information Systems (WIS), where HTML pages and WWW browsers are used to build the application interface. In this case, the read/write operations increase the traffic of data over the Web; this delay, in addition to the above referred inefficiency of the data transference operations between the central memory and the disk, justifies the previous statement.

SQL–the Structured Query Language, appeared in the early 70s (Codd, 1970; Chamberlin and Boyce, 1974) and ten years after became an international standard—ANSI, in 1986, and ISO, in 1987—for database querying and management.

SQL incorporates natural and powerful commands to: create tables, insert or update values into the record fields, select records (lines) or fields (columns) from a table, and join records of different tables. But more than a collection of database access operations, SQL is the reification of the relational data model paradigm. This is, using SQL strongly influences the rationality behind the program, the programming style.

Object-orientation completely changed the software development approaches, from problem analysis to programming style and techniques as well as program testing. Object-oriented approaches pose a new challenge in what concerns data persistency. Although there exist some object-oriented databases, the most usual is to stream object data and archive it into a relational database. Adopting this last policy, requires an *objectification layer*, to make that relational data available to the OO programs at the business or interfaces layers.

Recently, Microsoft came out with LINQ, the .NET Language Integrated Query. LINQ is supposed to be a natural extension to C# or Visual Basic to integrated the access to relational databases with those programming languages commonly used in the .NET framework.

This paper is concerned with the performance of C# when using LINQ, comparing this approach against the previous ones based on SQL or Objects.

Moreover, we also discuss wether LINQ extension to C# introduces or not any shift in the programming paradigm We reason about the previous and actual way of planning (or schematize) the program, in what concerns database access, to understand if use of Linq induces a new way of thinking, this is a new program-

da Cruz D. and Rangel Henriques P. (2009).
ASSESSING DATABASES IN .NET - Comparing Approaches.
In *Proceedings of the 11th International Conference on Enterprise Information Systems - Databases and Information Systems Integration*, pages 278-282
DOI: 10.5220/0001953402780282

ming style.

To attain these objectives, we compare the approaches that can be followed in the .NET framework to access a database and also the methods provided by .NET along Section 3. To guide this comparison, a running-example (introduced in Section 2) will be used. Section 4 closes the paper with some remarks.

2 RUNNING EXAMPLE

The example that we would consider in this paper is related with an Information System that we develop one year ago (before the latest stable version of LINQ) in the context of promoting the Portugal's northern region.

This information system, called SIGON.2, is aimed at supporting applicants (private companies or government institutions) in the the fulfillment of the application-forms that they should submit to apply for funds. Under that financial programme, an application is eligible if the funds demanded will be used to promote projects that contribute to the development of the northern region of Portugal.

Because the application-form is a complex object[1], its fulfillment can not be accomplished at once (actually this task can last days or even weeks). As a consequence, it should be possible to perform the task in several phases. However, no rule should be imposed, stating where or when these phases start or end; the applicant should be allowed to interrupt his work whenever he wants, and then continue in another moment without loosing data.

This statement led us to store (temporarily) the data fulfilled by each applicant in an intermediate object, a DataSet, that is then streamed into a line of a database table, as a XML document. After the complete fulfillment of the application-form, this data is convenient stored into the tables of a relational database. That is the core (the central component) of SIGON.2.

An application will contain one or more projects of the same type or of different types. A project can be one of the type: *immaterial* (when the contribution is at the knowledge level); *infra-structural* (as can be deduced from the name, this type is concerned with the development of infra-structures); and *mixed* (is a composition of projects that will contribute with both knowledge improvements and new infra-structures).

Over these undergoing applications, SIGON.2 should be capable of compute some statistics to display the type of each project. Besides its main functionality, SIGON.2 should be capable of computing statistics over the undergoing applications (those not yet closed and submitted, but under fulfillment) by project type (immaterial, infra-structural, mixed).

To compute these statistics it is necessary to traverse each XML's DataSet, search for the correct element, and count it.

As can be easily deduced, this task requires a considerable amount of time, for the reason that each XML reflects the structure of a complex application-form and stores all the fulfilled information.

Because of the great amount of time required to compute these statistics, we choose this example to show how it can be implemented using the data source controls referred previously, as well as, to compare the performance between them.

[1]Composed by a long list of interrelated items that can vary from a call to another, many of them being dynamic lists on tables.

3 COMPARING APPROACHES

In this section we compare the three approaches to access a database, by discussing the resolution of the practical case-study introduced in Section 2. Besides the methodological discussion, we will also measure their performance.

To implement the SIGON.2 statistical feature, as described in Section 2, all that is necessary is to perform a traversal to each XML document (that represents an application-form) and test if it is of type Infra-structural, Immaterial or both, to increment the respective counter.

3.1 LINQ Approach

As the data we want to access is stored in a relational database, the first thing that we need to prepare before being able to use LINQ with SQL is a data context class (named `DataContext`). The purpose of this class is to translate requests for objects into SQL queries addressed to the database and then assemble objects out of the results. The data context provides access to the tables in the database.

Essentially, the data context class maps database tables into classes, table's columns in properties, and relationships between tables are represented by additional properties.

This class can be automatically generated using the graphical LINQ to SQL Designer tool (provided by Microsoft Visual Studio 2008 (Powers and Snell, 2008)) or using the command-line SqlMetal tool (Microsoft, 2009).

Table 1 displays the code that implements the statistical feature.

Table 1: Computation of statistics using LINQ.

```
SigonDataContext sigon = new SigonDataContext();

var applications = from r in sigon.DataSetTemporarios
                   select r.dataSet;

var elementsInfra = from application in applications.ToList()
                    where application.Elements("infra-structural").Count() > 0
                       && application.Elements("immaterial").Count() == 0
                    select application;

var elementsIma = from application in applications.ToList()
                  where application.Elements("infra-structural").Count() == 0
                     && application.Elements("immaterial").Count() > 0
                  select application;

var elementsMix = from application in applications.ToList()
                  where application.Elements("infra-structural").Count() > 0
                     && application.Elements("immaterial").Count() > 0
                  select application;

countInfra = elementsInfra.Count();
countImma  = elementsImma.Count();
countMix   = elementsMix.Count();
```

As can be seen, this implementation uses both LINQ to SQL (first query) and LINQ to XML (other 3 queries).

With respect to LINQ to XML provider, it extends the language-integrated query features offered by LINQ to add support for XML. It offers the expressive power of XPath and XQuery but embedded in the programming language of our choice, with type safety and IntelliSense.

Looking carefully into the code of the Table 1, we observe that the first retrieves from the database all the XML into a list. After that, we separate the list of applications into three sublists by type, querying the XML documents in the same way (with the same syntax) that we query the database.

One of the first advantages of LINQ, that stands out from this example, is the expressiveness it offers: we can express declaratively what we want to achieve using queries instead of writing convoluted pieces of code.

3.2 SQL Approach

Table 2 shows the typical way to access a database in a .NET program.

In this approach, we typically start by defining a `ConnectionString` that contains information with respect to the database that we want to access.

After that, the *select* command to retrieve data is hard-coded (written explicitly in the source code). Then, we can go through the retrieved data using the specific API' provided by .NET framework (e.g., the `SqlDataReader`).

Inspecting carefully the code in Table 2, we can list several limitations of this approach:

- Although we want to perform a simple task, several steps and verbose code are required;
- Queries are expressed as quoted strings, which means that no compile-time checking is available. Then, some questions raise up: what if the string does not contain valid queries? And if a column is renamed in the database?
- The query results are loosely defined w.r.t. type (always retrieved as `Strings`).
- The classes we are using are dedicated to SQL Server and cannot be used with another database server.

Usually, this approach is used for RAD[2] or Prototyping.

3.3 Objectification Approach

Table 3 shows the implementation of the statistical feature using objects in the Data Access Layer (DAL), in a 3-tier approach.

The first thing that is necessary is to define a set of classes (objects) to deal with the database. These objects define the connection mode to the database (similarly to the `ConnectionString` referred above) as well as a set of methods that enable us to query the database.

`DataSetTemporarioDAO` (in Table 2) class specifies how we connect to the database; similarly, `DataSetTemporario` defines the referred methods to retrieve data.

The main advantage of this approach is that business logic can be defined once within the business layer and then shared with all the components within

[2]Rapid Application Development

Table 2: Computation of statistics using SQL.

```
string connectionStr = ConfigurationManager.
           ConnectionStrings["SigonConnectionString"].ConnectionString;
int countInfra = 0, countIma = 0, countMix = 0;

using (SqlConnection connection = new SqlConnection(connectionStr))
{
  connection.Open();
  SqlCommand command = connection.CreateCommand();
  command.CommandText = @"SELECT dataset FROM DataSetTemporario";

  using (SqlDataReader reader = command.ExecuteReader()) {
    while (reader.Read()) {
       string xmlContents = reader.GetString(0);
       XElement element = XElement.Parse(xmlContents);
       countInfra += (element.Elements("infra-structural").Count() > 0
                   && element.Elements("immaterial").Count() == 0) ? 1 : 0;
       countIma   += (element.Elements("infra-structural").Count() == 0
                   && element.Elements("immaterial").Count() > 0) ? 1 : 0;
       countMix += (element.Elements("infra-structural").Count() != 0
                && element.Elements("immaterial").Count() != 0) ? 1 : 0;
    }
   }
}
```

Table 3: Computation of statistics using Objects.

```
Sigon.DAL.DAO.DataSetTemporarioDAO datasetDAO = new DataSetTemporarioDAO();
List<Sigon.DAL.DataSetTemporario> datasets = datasetDAO.SelectAll();
int countInfra = 0, countIma = 0, countMix = 0;

foreach (Sigon.DAL.DataSetTemporario d in datasets) {
   StringWriter sw = new StringWriter();
   d.DataSet.Save(sw);
   XElement element = XElement.Parse(sw.ToString());

   countInfra += (element.Elements("infra-structural").Count() > 0
              && element.Elements("immaterial").Count() == 0) ? 1 : 0;
   countIma   += (element.Elements("infra-structural").Count() == 0
              && element.Elements("immaterial").Count() > 0) ? 1 : 0;
   countMix   += (element.Elements("infra-structural").Count() != 0
              && element.Elements("immaterial").Count() != 0) ? 1 : 0;
}
```

the presentation layer (Sheriff, 2002). Any changes to business rules can therefore be made in one place and be instantly available throughout the whole application.

The strongest reason to use this approach is reuse: logic placed in a business layer increases the reusability of an application. It is possible to change the contents of any one of tiers (layers) without having to make corresponding changes in any of the others. It also enables parallel development of the different tiers of the application.

However, there some disadvantages using this approach: there is more processing on the web server; a more complex structure is involved; it is more difficult to setup and maintain; and the physical separation of application servers (containing business logic functions) and database servers (containing databases) may moderately affect performance.

Even if we use a code generator or one of the several object-relational mapping tools available, the generated code has its own limitations. For instance, that code is designed for accessing databases, and that code do not deal with other data sources such as XML. Also, that code do not integrate data-access and data-querying features right into the language of our choice. So, mapping tools are a partial solution for the problem.

3.4 Measuring the Performance

In this subsection, we present the execution time when running the three variants of the case-study over a database with 1105 Application-forms (Table 4, column Q1), in order to rank the three approaches.

To sustain the conclusion, we also implement other three queries, coded according to the three approaches under comparison. The execution times observed are also registered in Table 4.

The new queries are:

Q2 Retrieve and display all the partners of an Application promoter (this problem requires the interconnection of 3 tables);

Q3 Retrieve and display the name of all the Application-forms;

Table 4: Computation times to Queries 1-4.

	Q1	Q2	Q3	Q4
LINQ	21,108	0,022	0,036	0,18
SQL	5,981	0,001	0,029	0,439
Object	87,274	0,008	0,631	2,071

Q4 Retrieve and display all the projects included in all Applications;

To display the retrieved results, we used the data-bound controls referred previously (`GridView` and `FormView`).

As can be deduced from Table 4, SQL is the fastest. The objectification is the second. And the last is the LINQ.

However, combining the objectification approach with the features of LINQ we obtain much better results, concerning `Q1`: 10,986 seconds — a reduction of 78%.

4 CONCLUSIONS

At present moment, before starting the development of a new software project, the programmer or the software engineer should invest a lot of effort in selecting the appropriate working environment. After choosing the programming framework, he will face other challenges that deserve some deep work: it is important to be aware of the new technological trends, but it is also mandatory to understand which of them constitute a true paradigm shift, or those that are just new technical resources.

As reported along the paper, we tackled some experiments to measure the actual impact of using LINQ in C# programs, concerning execution time, when compared with SQL or *Objectification* classical approaches; we also studied the impact in the programming style. As we foresaw, the use of LINQ implies a new style of include in C# data access operations; it is more high level, more natural and let us check the correctness of the statements at compile time. Of course, the tradeoff is the execution time, that is much bigger. Analyzing the three compared approaches from a compilers perspective, it was not difficult to forecast the ranking obtained with SQL—the *assembly* of database engines—in the first position (the faster), and LINQ in the last position (the slower). Concerning the *Objectification* approach, it occupies the second pole position, closer to SQL or LINQ depending on the style adopted; if we combine it with `XElement(LINQ)` its performance is not sp far of that obtained with SQL.

To conclude, we would say that SQL approach—completely tuned for extracting information just from databases—should be used for RAD[3] or Prototyping. It is dependent on the data source and is similar to hard-code the select command in the high-level C# oo-programs. For a 3-tiers programming approach, in classical object-oriented style with a DAO in the DAL, the second approach is a good one, and is the one we recommend if efficiency is the concern. The use of LINQ clearly proved to the be more declarative and better style, keeping the source code independent of the data source, and should be used every time that execution performance is not critical.

REFERENCES

Chamberlin, D. D. and Boyce, R. F. (1974). Sequel: A structured english query language. In *FIDET '74: Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264, New York, NY, USA. ACM.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387.

Microsoft (2009). Code generation tool. `http://msdn.microsoft.com/en-us/library/bb386987.aspx`.

Powers, L. and Snell, M. (2008). *Microsoft®visual studio 2008 unleashed*. Sams, Indianapolis, IN, USA.

Sheriff, P. D. (2002). Designing a .net application.

[3]Rapid Application Development