The 16th IASTED International Conference on Software Engineering and Applications (SEA 2012)
November 12 – 14, 2012. Las Vegas, USA

DATA MIGRATION AND VALIDATION USING THE SMART PERSISTENCE LAYER 2.0

Mariusz Trzaska
Polish Japanese Institute of Information Technology
Koszykowa 86, 02-008 Warsaw, Poland
mtrzaska@pjwstk.edu.pl

ABSTRACT
We present a new version of the Smart Persistence Layer (SPLv2) which is a working prototype following our approach to modern data sources. Contrary to the most popular solutions, it does not use Object-Relational Mappers nor relational databases. This approach guarantees complete lack of the impedance mismatch. The new features of the SPLv2 concern data migrations and validations as an answer to real-world projects where models are changing all the time. Moreover, the conducted comparative benchmarks (db4o, Perst, MS Entity Framework) prove usefulness of our approach. The prototype, together with previously existing functionalities (like transparent persistency, bidirectional associations) makes an interesting alternative to existing data sources.

KEY WORDS
Impedance mismatch; Databases mapping; Object-Relational Mappers; ORMs; Persistence; LINQ.

1. Introduction

The initial release of the Smart Persistence Layer (SPL) [1] has been designed and developed as a solution to the impedance mismatch problem. The second version of the tool introduces new important features, but still reassembles the only valid approach to solve the problems caused by the mix of object-oriented and relational models. Before discussing the mentioned new features, we would like present a short description of our motivation: the impedance mismatch problem.

The modern software, especially the web one, is usually created using two models: object-oriented and relational. The first one is utilized in nowadays programming languages (Java, MS C#, Ruby, etc.), where most of the business logic is implemented. The latter is involved in a data layer. The programmer has to deal with two models, transforming one to another. To make it easier, Object-Relational Mappers (ORMs) have been developed. They map constructs from one model to another one, mostly using some additional information provided by the programmer. Unfortunately, even best ORMs are not able to transparently separate a user from a data's relational model. The main reason of that is the fact that probably there is no general algorithm that maps object-oriented queries and updates into SQL and still ensures good performance and flexibility. In fact, it does not even matter how the mapping is to be defined: using a configuration file, a DSL or some other way. The result is still the same: the programmer has to spend his/her time doing some repetitious and error-prone work.

This uncomfortable situation is caused by relational database management systems which have currently a dominant market share. Discussing the reasons behind the situation is beyond the scope of this paper, but could be found in [2].

Existing mappers, even those evolving for years, still require significant attention from programmers. In our opinion, this proves that the current approach has serious limitations which probably will never be overcome. Thus, the only promising solution is to use a single model both for a programming language and an utilized data source. Because of its flexibility and power we believe that we should use object-oriented model.

The rest of the paper is organized as follows. To fully understand our motivation and approach some related solutions are presented in Section 2. Section 3 briefly discusses key and new concepts of our proposal and its implementation. Section 4 contains comparative benchmarks. Section 5 concludes.

2. Related Solutions

In this section we would like to present solutions related to our prototype on two different levels. The first one describes various ways of dealing with the impedance mismatch and the second one discusses approaches to new features introduced by the new version of the SPL.

2.1 The Impedance Mismatch

There are different approaches to the impedance mismatch problem. Some of them try to solve the problem by extending programming languages with declarative specification capabilities like JML [3] or Spec# [4]. Generally we do not accept such solutions mainly because of the complexity, e.g., Spec# requires a dedicated compiler.

Another method is based on using an object-oriented database management system (ODBMS) which has an object model compatible to a chosen programming

language, e.g., db4o [5, 6] or Objectivity [7]. Both of them are mature solutions existing on the market for at least 10 years. However in some cases, using them could be too complicated. For instance, the Objectivity/DB requires dedicated class modeler, which generates classes containing a special code. Therefore, it is not possible to use simple POCOs (Plain Old CLR Objects). This term means utilizing classes without the necessity of inheriting from dedicated super classes or implementing particular interfaces. Similar approach is employed in Perst [8]. In order to achieve maximum performance it requires implementing a dedicated interface. Furthermore, there is no automatic extent management, which has to be implemented manually as a persisted, root object. On contrary, both db4o and our prototype do not impose such limitations.

The reference [9] provides the list of open source persistence frameworks for the MS .NET platform. Unfortunately, most of them are implemented as ORMs, which of course introduce some level of impedance mismatch. We have found only two tools, which do not utilize a relational database: Bamboo.Prevalence [10] and Sisyphus [11]. However they usually require some special approaches, e.g., a command pattern utilized for data manipulation for the Bamboo and the necessity of inheritance from a special class for the Sisyphus.

Developing an application using a platform containing a merged database and a full programming language is another way of dealing with the impedance mismatch. However this kind of solutions are pretty rare, mainly because of an insufficient number of decent tools. Admittedly there are various DBMS supported by languages, e.g., T-SQL, PL/SQL but they are utilized usually with a “real” programming languages like Java, C# or Ruby. These DB languages have imperative functionality and PL/SQL has even some object-oriented constructs. There are also fully object-oriented solutions like SBQL for the ODRA platform [12]. These seem more appropriate thanks to the more powerful and flexible model.

Discussing related solutions, it is worth mentioning the NoSQL family of platforms. There are no strict definitions, but they usually do not have a fixed schema/model nor the support for SQL. In various cases ACID transactions functionality is also absent. Some of them store their data as key-values pairs or documents. Despite not the best opinion in the research community (e.g. because of a limited data model), they prove their usefulness in well-known commercial projects like Twitter, Amazon or Google. The most prominent are Apache CouchDB [13], MongoDB [14], Berkeley DB [15]. The utilized way of storing information, e.g. key-value pairs, reduces the DB model to very simple attributes which could be easily stored in a NoSQL database. By its simplicity, the model becomes almost transparent to a programmer substantially reducing the impedance mismatch.

Thus, our proposal is based on replacing both an ORM and a database with a data source native to a programming language. As a result, there is no impedance mismatch at all. The approach is supported by a working prototype for the .NET platform. The prototype provides a persistence layer and extent management for objects of a programming language. Furthermore, in version 2.0, we have added support for data migration and validation.

2.2 Data Migration and Validation

Data migration is a very important aspect of every real-world application. It is very hard to develop software which stood without changes for years. Usually changes affect all layers of an application, including its data model. When users provide some data, and the model changes, there is a problem of data migration. It is crucial to apply changes in a way retaining all the existing data.

Typical relational databases usually deal with the issue using SQL and a three steps procedure:

- Export existing data to a textual file,
- Apply model changes,
- Import previously exported data to the format implied by the new model.

The procedure is straightforward only if one modifies the model by adding some properties without removing/changing existing ones. Otherwise this could be really complicated and might require dedicated manual activities (e.g. intermediate transformations) and/or additional tools. For instance, [16] compares database schemas (before and after applying the changes) and deploys differences. More general information regarding this topic could be found in [17].

Besides databases such migrations affect also Object-Relational Mappers. The new release of the Microsoft Entity Framework (MS EF) [18] introduces a support for data migrations. The process is triggered by a programmer and scans the model for changes. Then, any pending change is applied to the database, or a special SQL script is generated. It is also possible to customize the transformation manually, using a dedicated API.

The db4o handles simple schema changes automatically. Adding a new class attribute does not interfere with reading existing values. Removing an attribute just ignores its values in the stored data. More advanced changes are performed via a special API. The Objectivity/DB has similar features.

Our approach (SPL v2) to handling data migrations is described in Section 3.2.

Another important aspect of modern data management is validation. The purpose of this activity is to guarantee conformation to the business rules of the modeled world. In case of relational DBMS, the most common approach uses constraints (e.g. required length, text input using regular expressions, uniqueness or number ranges) and/or triggers which can handle more complicated cases.

MS EF performs data validation based on model’s annotations. A programmer decorates particular model’s elements. Another way offering similar capabilities is to

use a special API. The result is that only validated objects can be saved to the data source (usually a relational DBMS).

Unfortunately, the object-oriented database management system Objectivity/DB does not support data validation at all. Thus the process has to be implemented manually outside the tool. According to [19] (page 258) db4o only allows for checking uniqueness of field values. However it is possible to utilize standard .NET data annotations (used also by MS EF) together with third-party providers (e.g. [20]) and some additional configuration code. In our opinion, this approach looks a bit complicated. Our SPL follows another convention (see section 3.3).

3. The Smart Persistence Layer 2.0

One of the strongest motivations for using a database management system is a query language. Probably the most popular one for a .NET platform is LINQ [21] available in many flavors. Basically, the LINQ adds powerful query capabilities [22] to ordinary programming languages (e.g., C# and Visual Basic). The LINQ works with native collections of the programming language allowing querying them as regular databases. It is also supported by some "real" databases and various ORM mappers including their own solution called Entity Framework [18]. Generally speaking, the mapper uses a relational database for storing data which, of course, causes some impedance mismatch (especially concerning inheritance).

In real-world business applications data persistency is strongly required. Unfortunately the .NET platform does not provide such functionality. We do not take into account the serialization mechanism because it stores the entire graph of objects every time. To fill the gap we have designed and developed our prototype as a transparent complement of the query language. Such an approach guarantees that every kind of impedance mismatch simply disappears. Furthermore we do not want to make programmers use any kind of super classes or implementing special interfaces (POCO objects are good enough).

The way we designed the prototype makes it possible to implement it for other platforms with the reflection capabilities, e.g., Java. In this case it would be possible to reuse significant parts of the source code and data files as well.

3.1 Class Extents and Bidirectional Associations

This section contains only a short overview of the implemented functionality with emphasis on changes. Detailed information could be found in [1].

The most basic functionality for a data source is delivering an extent of objects belonging to a particular class. This could be achieved using many ways. For instance the db4o [6] uses the following code:

```
IList <Pilot> pilots =
      db.Query<Pilot>(typeof(Pilot));
```

However in our prototype we have simplified that to:

```
IQueryable<Pilot> pilots =
              db.GetExtent<Pilot>();
```

Please note that our method does not require the parameter, but the result is still strongly typed.

Another area related to an extent, which needs a clarification is how and when new objects will be incorporated into extent. We have decided that every object made persistent will be added to an extent. It is also possible to manually execute a dedicated method. Of course, if a programmer would like to achieve automatic adding to an extent, then the method could be executed in a constructor of a class.

One of the key functionality of every data store is the ability for creating and persisting connections among objects. In our opinion, it is especially useful if the connections are bidirectional allowing navigation in both directions (e.g., from a product to its company and vice versa). Unfortunately, databases usually do not support the feature. According to [6] the db4o does not have it either. This is also the case of native references existing in popular programming languages (e.g., MS C#).

The implementation of the mentioned functionality is complicated especially if we would like to work with the POCO (Plain Old CLR Object) objects. This approach means that we cannot expect implementing a specified interface or functionality inherited from a super class. Another disadvantage of putting links into a super class would be problems with navigation using the LINQ.

One of the approaches is generating classes based on some templates. This is the case of one of the options in the Microsoft Entity Framework [18] and Objectivity/DB [7]. However, this functionality requires some kind of support from a tool and in our opinion may not be useful for all programmers.

As an another improvement, the new SPLv2 uses the following simple code for creating a bidirectional association:

```
ICollection<Tag> Tags = new
            SplLinks<Tag, Product>(t =>
            t.Products, this);
```

Please note that the entire construct is strongly typed and uses a lambda expression to define the reverse attribute name. Such a solution eliminates syntax errors and adds IDE hints. The `SplLinks` class implements ordinary .NET interface for accessing collections thus using it is exactly the same as any other .NET collection. Creating a bidirectional link requires only executing a single Add method with the target object. The reverse connection will be created automatically based on previously defined data. Of course, all LINQ queries work as well.

3.2 The Transparent Persistence and Data Migrations

Comparing to the previous release of the SPL, our persistence mechanism has undergone some changes. They have been mainly caused by the new data migration feature which supports the following cases (all of them occurs after persisting some objects):

- Adding a new attribute to the class/model. The read object will have a default value of the new attribute and previous values of the unchanged attributes;
- Removing an existing attribute. In this situation, the object will be instantiated with existing attributes' values (the non-existing ones will be ignored);
- Renaming an attribute. This cannot be handled automatically but could be resolved using special rules, e.g. the Product class had an attribute named Price which has been renamed to TotalPrice (please note that the entire expression is strongly typed wherever possible):

```
db.AddMigrationRule<Product>("Price", p => p.TotalPrice);
```

During saving objects the following data has to be persisted:

- business content of the objects,
- location of the above,
- information about their types (classes),
- information about attributes.

All of them can change and grow during the run-time. We have decided to use two files: the first one will hold business information whereas the second one some technical details. Initially we thought about three files but the types information is usually quite small and repeatable thus can be stored at the beginning of the second file – after the header (Figure 1). A programmer can define amount of the allocated space for the purpose. A default value is 1MB, which makes possible storing about 3000 entries. It is possible to use just one file but at cost of more complicated design and possibly worse performance.

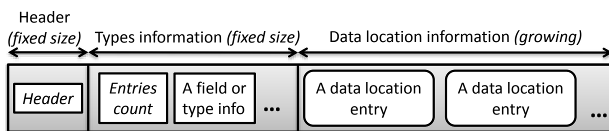

Figure 1. Structure of the file storing types and location information

The single entry regarding the location of data (the data location entry from Figure 1) consists of:

- object identifier;
- identifier of its type;
- location in the data file where the object's content starts. This entry is updated every time when an object is saved;
- location in the index file where the location data starts.

The above information also exists in the memory to boost performance. It is saved to disk only as a backup and for reading objects purposes.

As mentioned previously we do not persist classes (types) in the file. Thus during an object initialization those classes have to be accessible by the .NET run-time (e.g., as standard DLL libraries). This rule does not apply to attributes.

The other file, with business data of persisted objects, can be read only using the location and types information. It is read at the very beginning. The current prototype reads all data to the memory. This could be a problem in some cases but modern computers are usually equipped with a lot of RAM. See also Section 4 for various benchmark results.

The process of saving and reading objects intensively uses the reflection mechanism. Currently it is able to deal with atomic types, lists (classes implementing the `IList` interface), `ICollection` and other types built using these invariants (see also the sample in Section 4).

One of the problems related with links, which should be addressed, is persisting connected objects. When we would like to persist an object, how should we act with all referenced objects? There are different approaches, e.g., db4o [7] uses a concept called update depth. This is simply a number telling how many levels of connections should be saved. We have decided to follow another approach. When we save an object, all referenced unknown (not saved previously) objects are saved, no matter how deep they are. Thus the first execution could be costly, but the objects have to be saved anyway. All next updates will not save known objects. If a programmer wants to save them, then it has to be done directly by executing the `Save` method. The method should also be utilized every time a single object is modified (its content will be persisted in the file). This policy guarantees that persisting objects will not be costly.

3.3 Data Validation

The SPLv2 supports data validation using custom annotations [23]. Attributes in the model can be decorated with various markers. Then a programmer can execute a single method which will return errors or null otherwise.
The library is shipped with some validators, namely: e-mail, number range, regular expression, and required. It is also possible to easily create a custom validator by implementing a single method:

```
public abstract string Validate(object
    attributeValue, Type attributeType,
    string attributeName);
```

The method's body receives a current value of the attribute being checked, its type and name. The method should return a description of the problem or null if the attribute has been validated successfully. Of course a

programmer can implement the method anyway he/she wants.

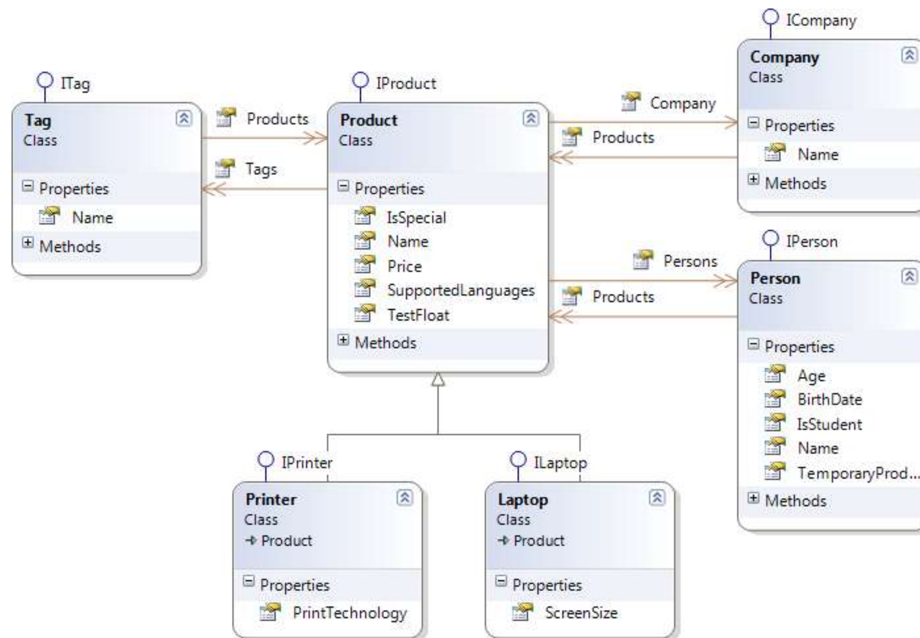

Figure 2. A class diagram of the sample utilized to the comparative benchmarks

4 The Comparative Benchmarks

To verify usefulness of our prototype we have decided to perform some benchmarks. On contrary to the previous release [1], this time we have executed some comparative tests with the following solutions:

- The Smart Persistence Layer v2,
- Db4o v8.0
- MS Entity Framework 4.3 Code First + MS SQL Server Express
- MS Entity Framework 4.3 Code First + MS SQL Server Compact
- Perst v4.32

Benchmarking and comparing different tools is always a challenge. It is not an easy task to assure that the tests will be impartial and will not favor one solution over another. To achieve this we have decided that all tests for all platforms will be performed by the same program. This became possible by designing common interfaces implemented by particular solutions. The interfaces define two kind of information: a business model and benchmarks itself.

The business model utilized for the test is presented on Figure 2. It could be summarized as follow:

- Products have various properties including: a name, a price and a list of supported languages;
- Every product can be described using various tags;
- A company manufactures many products, but a product is related to a single company;
- A product is supervised by many persons and a single person can supervise many products;
- There are various kinds of products with different properties. Printers contain information about utilized print technology and laptops store a screen size.

Although the presented case is quite simple, it contains different kinds of business information. Thus it allows comparing different data sources in common activities.

The diagram from Figure 3 shows structure of the program executing the tests. The base class `PerformanceBase` contains all necessary methods. However, some of them are abstracts and have to be implemented in particular subclasses. Basically all of them are responsible for opening a data source, generating some data and validating it by executing test queries. Different kind of tests inherits from the base class. For instance the `PerformanceTest2` class, besides generating necessary data, defines the following activities (all of them utilize LINQ to query data):

- Check the products count,
- Find a product having particular tag and not being a laptop,
- Find a product having particular tag and not being a printer,
- Find a product of particular company and having specific number of tags,
- Find a product supported by a given number of languages,
- Validate a person, who supervised a product,
- Count laptops,
- Find a laptop with specific tag,
- Verify manufacturer of a laptop,
- Count printers,

- Validate a printer's company,
- Count persons,
- Check number of supervised product by a particular person,

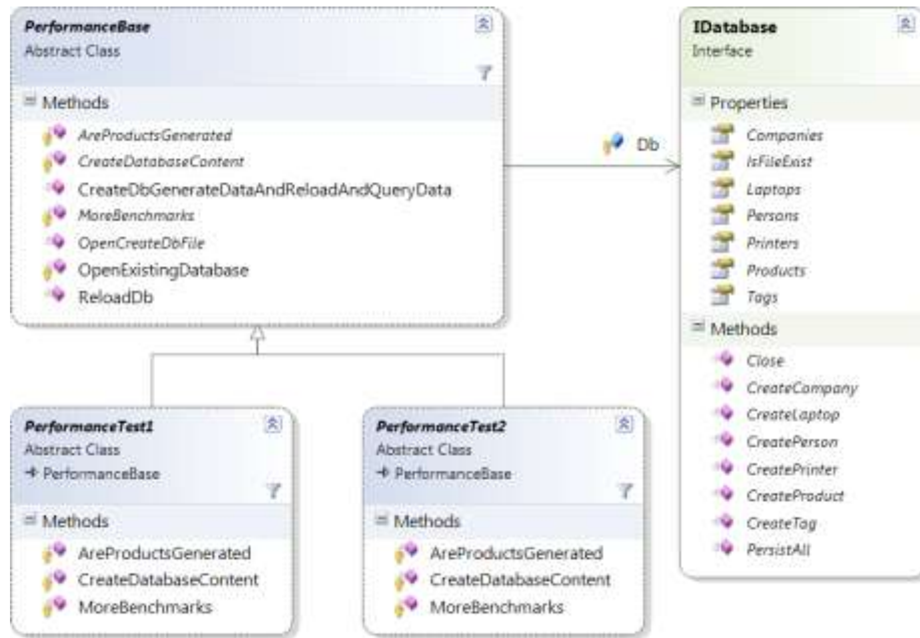

Figure 3. Class diagram for benchmarks structure

- Find youngest person,
- Count laptops with a particular price.

As mentioned before, due to the flexible architecture, all tests are executed by the same program against different data sources and various amounts of data. A single data source is accessible through the `IDatabase` interface (see Figure 3). The interface declares all properties (e.g. class extents) and methods (e.g. for creating objects) necessary for working with various data sources.

Each benchmark started with generating and persisting various amounts of data (see Table 1). Then the generated data has been validated by executing previously mentioned queries (Table 2).

After that a data source has been closed and all data removed from the memory. Next the data source, containing the generated data, has been opened. In case of the SPL2 it also meant reading all objects to the computer's memory (see Table 3) causing quite long times. Fortunately it is performed only once, when a program opens the SPL2.

Table 1. Generating and persisting data [s] (less is better)

Total objects	SPL v2	DB4o v8	Perst V4.36	EF CF 4.3 + SQL Server CE	EF CF 4.3 + SQL Server Express
48,000	14.67	7.50	***0.42***	6,607.96	5,169.50
330,000	98.86	54.75	***2.30***	*cancelled*	*cancelled*
1,650,000	465.85	319.62	***10.80***	*cancelled*	*cancelled*

Table 2. Validating generated data [s] (less is better)

Total objects	SPL v2	DB4o v8	Perst V4.36	EF CF 4.3 + SQL Server CE	EF CF 4.3 + SQL Server Express
48,000	***0.22***	25.09	0.44	0.74	0.48
330,000	***1.35***	183.19	2.80	*cancelled*	*cancelled*
1,650,000	***6.37***	916.37	13.17	*cancelled*	*cancelled*

Finally, with an opened data source, the data has been validated using the same queries (see Table 4).

Table 3. Opening a data source and optionally reading all objects [s] (less is better)

Total objects	SPL v2	DB4o v8	Perst V4.36	EF CF 4.3 + SQL Server CE	EF CF 4.3 + SQL Server Express

48,000	16.57	0.05	0.04	***0***	***0***
330,000	112.21	***0.05***	0.13	*cancelled*	*cancelled*
1,650,000	545.42	***0.25***	0.84	*cancelled*	*cancelled*

Table 4. Validating data (after reading from a data source) [s] (less is better)

Total objects	SPL v2	DB4o v8	Perst V4.36	EF CF 4.3 + SQL Server CE	EF CF 4.3 + SQL Server Express
48,000	***0.16***	27.61	0.74	40.36	5.42
330,000	***1.29***	205.08	5.62	*cancelled*	*cancelled*
1,650,000	***6.58***	1,095.26	26.62	*cancelled*	*cancelled*

As it can be seen from Tables 2 and 4 queries times vary significantly. The worst results have been achieved with the MS EF (both with the server's small edition (Compact) and the "real" one). This could be caused by the necessity of translating queries to SQL. Such translation is not always possible, e.g. when a programmer would like to query against a method (e.g. find all people older than 30 years; when an age is calculated by a property or a method). The same problem could be partially caused the way we have benchmarked (common interfaces, methods, etc.). In such cases, the mapper throws an exception saying that is not able to translate the method/property. The result is that a programmer needs to convert the extent to a list (which instantiates all objects) and query the list without SQL optimizations. However, we have no clue why generating objects (Table 1) by MS EF took so long.

The best results in querying were achieved by our prototype (SPLv2) due to the way it works. All data is kept in the memory which boost all queries but for the price of reading them at the start (Table 3). The remaining systems do not read the data at the beginning thus the opening times are really tiny.

It also could be interesting to compare files' sizes storing exactly the same (generated) data (see Table 5).

Table 5. Files' sizes storing the same (generated) data [KB] (less is better)

Total objects	SPL v2	DB4o v8	Perst V4.36	EF CF 4.3 + SQL Server CE	EF CF 4.3 + SQL Server Express
48,000	***5,706***	11,100	7,168	6,420	NA
330,000	***34,358***	80,794	48,624	*cancelled*	*cancelled*
1,650,000	***169,574***	444,217	237,826	*cancelled*	*cancelled*

Table 6. Detailed information about numbers of generated objects

Total objects	Products	Tags	Persons	Companies
48,000	7,000	35,000	3,000	3,000
330,000	70,000	200,000	30,000	30,000
1,650,000	350,000	1,000,000	150,000	150,000

Table 6 presents detailed information about numbers of generated objects.

Some other remarks regarding the procedure and the results are enumerated below:

- The test computer configuration: Intel Core i7 2.93GHz, RAM: 8GB, Windows7 x64;
- Due to significantly long execution times for the MS Entity Framework we have decided to perform tests only for the smallest extents;
- On contrary to the SPLv2, the db4o and Perst do not have native bidirectional associations. Thus they have been created manually;
- The Perst [8] system does not follow the pure POCO approach. According to the tutorial, the best results can be achieved when all model classes implement a dedicated interface or extends a provided class. We employed the second solution (a common super class). But yet the Perst performance surprised us in a very nice way.

The benchmark results clearly prove that even significant number of objects (more than one and a half million) could be processed by storing them all in a memory on a modern computer. This solution guarantees the best query performance outperforming the second place by 3-4 times (Table 4).

5 The Conclusion and Future Work

We have presented our approach to working with data which is supported by the new version of the Smart Persistence Layer (SPLv2). The SPLv2 together with an existing query language (LINQ) could be an interesting alternative to existing data sources (object-relational mappers or object databases). Of course it requires a lot of

work to leave a prototype stage and compete with well-established tools. However, during the comparative benchmarks proved that, despite storing all data in the computer's memory (1.5m+ objects), is able to work efficiently wining the query tests.

The contribution of this paper is based on new functionalities delivered by the SPLv2, namely: data migrations and validation. They make easier developing real-life systems, where model still evaluates and existing data needs to be compatible with a future meta data. The validation feature introduces an easy way for verifying objects using existing or custom annotations.

Another valuable input of the paper is comparative benchmarks with popular solutions (db4o, Perst, Microsoft Entity Framework). It seems that in general existing modern tools outperformed tested Object-Relational Mapper. Moreover the mapper (similarly to other ORMs), trying to reduce the impedance mismatch, still is not able to hide the relational database (e.g. queries against objects' methods/properties).

We hope that researching area of alternatives to existing relational databases we will be able to promote new approaches reducing or eliminating the impedance mismatch.

References

[1] Trzaska M.: The Smart Persistence Layer. ICSEA 2011: The Sixth International Conference on Software Engineering Advances. October 23-29, 2011 - Barcelona, Spain. ISBN: 978-1-61208-165-6. pp. 206-212.

[2] Objectivity, Inc.: Hitting The Relational Wall. http://www.objectivity.com/pages/object-oriented-database-vs-relational-database/default.html. Last accessed: 2012-06-15.

[3] Chalin, P., R. Kiniry, J., T. Leavens, G., and Erik Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In Formal Methods for Compo-nents and Objects (FMCO) 2005, Revised Lectures, pages 342-363. Volume 4111 of Lec-ture Notes in Computer Science, Springer Verlag, 2006, pp. 342-363

[4] Barnett, M., Rustan K., Leino M., and Schulte W.: The Spec# programming system: An overview. In CASSIS 2004, LNCS vol. 3362, Springer, 2004, pp. 144 – 152

[5] Paterson, J., Edlich, S., and Rning, H.: The Definitive Guide to Db4o. Springer (August 2008), ISBN: 978-1430213772

[6] db4o tutorial, http://developer.db4o.com/Documentation/Reference/db4o-8.0/net35/tutorial. Last accessed: 2012-06-15

[7] The Objectivity Database Management System. http://www.objectivity.com. Last accessed: 2012-06-16

[8] Perst - An open source, object-oriented embedded database. Last accessed: 2012-06-16

[9] Open Source Persistence Frameworks in C#. http://csharp-source.net/open-source/persistence. Last accessed: 2012-06-16

[10] Bamboo.Prevalence - a .NET object prevalence engine. http://bbooprevalence.sourceforge.net/. Last accessed: 2012-06-15

[11] Sisyphus Persistence Framework. http://sisyphuspf.sourceforge.net. Last accessed: 2012-06-16

[12] Adamus, R., Daczkowski, M., Habela, P., Kaczmarski K., Kowalski, T., Lentner, M., Pie-ciukiewicz, T., Stencel, K., Subieta, K., Trzaska, M., Wardziak, T., and Wiślicki, J.: Overview of the Project ODRA. Proceedings of the First International Conference on Object Databases, ICOODB 2008, Berlin 13-14 March 2008, ISBN 078-7399-412-9, pp. 179-197.

[13] Lennon J.: Beginning CouchDB. Apress, 1 edition (2009), ISBN: 1430272376.

[14] Plugge E., Hawkins T., Membrey P.: The Definitive Guide to MongoDB: The NoSQL Da-tabase for Cloud and Desktop Computing. Apress; 1 edition (2010). ISBN: 1430230517.

[15] Yadava H.: The Berkeley DB Book. Apress; 1 edition (2007). ISBN: 1590596722

[16] RedGate SQL Compare. http://www.red-gate.com/products/sql-development/sql-compare/. Last accessed: 2012-06-14

[17] Morris J.: Practical Data Migration. British Informatics Society Ltd (2006). ISBN: 1902505719

[18] Lerman, J.: Programming Entity Framework: Building Data Centric Apps with the ADO.NET Entity Framework. O'Reilly Media, Second Edition, ISBN: 978-0-596-80726-9 (2010)

[19] Versant Corporation: Db4o 8.1 Reference Manual. http://www.db4o.com/. Last accessed: 2012-06-16

[20] Hibernate Validator. http://www.hibernate.org/subprojects/validator.html. Last accessed: 2012-06-16

[21] Pialorsi P., Russo M.: Programming Microsoft LINQ in Microsoft .NET Framework 4. Microsoft Press, ISBN: 978-0735640573 (2010)

[22] 101 LINQ Samples. http://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b. Last accessed: 2012-06-16

[23] Creating Custom Attributes. http://msdn.microsoft.com/ en-us/library/sw480ze8(v=vs.100).aspx. Last accessed: 2012-09-26