STUDIA INFORMATICA 2009
Volume 30 Number 2A (83)

Andrzej SIKORSKI, Jarosław CIEMNOCZOŁOWSKI
Politechnika Poznańska, Instytut Automatyki i Inżynierii Informatycznej

DEFERRED EVALUATION OF PATH QUERIES ON HIERARCHICAL DATA

Summary. This paper is dealing with the on-line processing of queries on hierarchical data. We make use of the structural join method, that we modify so as to achieve the minimum number of I/O operations required for successful identification of the result set. Our technique along with the application server hosting model and distributed architecture allows efficient implementation of information systems extensively using hierarchical data.

Keywords: multi-tier architectures, distributed computing, XML, structural join

OPÓŹNIONE PRZETWARZANIE ZAPYTAŃ ŚCIEŻKOWYCH W HIERARCHICZNYCH STRUKTURACH DANYCH

Streszczenie. Artykuł dotyczy przetwarzania danych hierarchicznych w trybie bezpośredniej dostępności danych. Podstawą metody jest zmodyfikowany algorytm złączenia strukturalnego, minimalizujący liczbę wykonań operacji wejścia-wyjścia niezbędnych do pomyślnego znalezienia zbioru wynikowego. Zaproponowana technika umożliwia implementację wydajnych serwerów aplikacji efektywnie przetwarzających hierarchiczne zasoby danych.

Słowa kluczowe: architektury wielowarstwowe, przetwarzanie rozproszone, złączenie strukturalne

1. Introduction

Hierarchical data is ubiquitous and exist in many applications including web, business, ontology databases, expert systems. We tackle the optimization techniques for searching very large hierarchical data performed by application servers in the context of distributed information systems. We are going to exploit the hybrid operational characteristics of

application server, which is a processing node inherent to distributed architectures and belongs to the client layer, as far as a logical model is concerned, but technically is implemented as a server.

We claim that the hierarchical query engine can be implemented as an application server on the top of a suitable data manager, which apart from searching, indexing and data storage supports also low level concurrency and transactional processing. In fact, such an architecture could offer more flexibility as business rules may be seamlessly integrated into the low level query facilities (e.g. structural joins). This flexibility compares with, the one supported by stored procedures in SQL. However, let us keep in mind that native XML databases do not yet include support for application code embedding (i.e. triggers and stored procedures). We show that at low level our implementation provides a better co-operation with concurrency and transaction processing.

The conventional architecture of hierarchical data processing makes use of either database servers (XML, SQL native, relational, hybrid) or delegates the relevant tasks to some specialized query engines (c.f. [1], [2]). On the contrary, multi-tier architectures make available the opportunity enabling application code and data to share the same host. Due to this property extensive and costly data transfers can be avoided; this was once the main point about SQL, which is the special case of client server architecture. The same paradigm underlies a large variety of query engines including native XML databases.

Our research focuses on hierarchical structures, where data items are coupled by the parent-child and ancestor-descendant relations. The ancestor-descendant relation is understood as a path between two nodes consisting of parent-child edges. Fig. 1a) visualizes our presentation convention – a many nodes path is shown as a dashed line while direct parent-child edge is solid. This is the logical model, the label based physical implementation, is given in Section 1 (the label based technique is required for our purposes).

The nodes to be seen in Fig. 1 b) are a subset of a possibly much bigger tree (e.g. ontology databases, XML documents). This tree is subject to complex search operations. We intend to accelerate this type of search by using indexing and structured storage techniques (cluster indices). The available auxiliary indices allow a speedy reduction of candidate nodes, the ones likely to appear in the result set. Our composition of deferred operators ensures that each candidate node is accessed only once. This result can be considered optimal in terms of the performance measures given in [8].

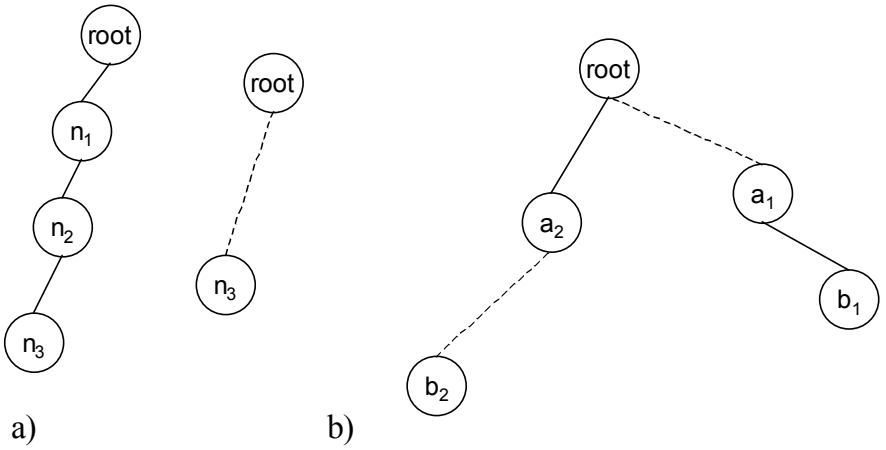

Fig. 1. Dashed line (a) shows a path between two nodes root and n_3. The root in (b) is a direct parent of a_2 and so is a_1 with respect to b_1. There are paths between root and a_1 and between a_2 and b_2

Rys. 1. Ścieżka między korzeniem a węzłem n_3 jest zaznaczona linią przerywaną (a). Korzeń jest rodzicem a_2 (b) , tak samo jak a_1 względem b_1. Korzeń i a_1 oraz a_2 i b_2 są połączone ścieżkami

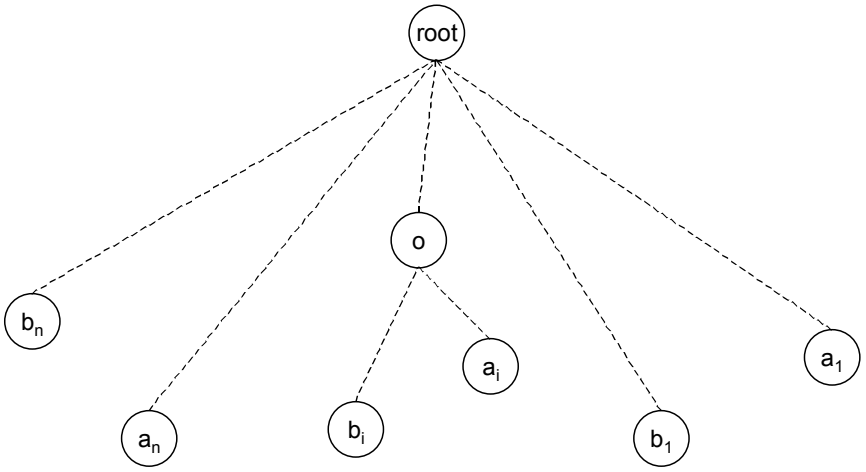

Fig. 2. There are two set of nodes (object[@class="a"], object[@class="b"]), determined by an index search on the data table. The result set for the *XPath* query /object[//object[@class-="a"] and //object[@class="b"]] can be obtained as a structural join performed on these datasets. The nodes labeled root and o are members of this result set

Rys. 2. Dwa zbiory (object[@class="a"], object[@class="b"]) wyznaczone za pomocą indeksu wchodzą w skład zbioru wynikowego będącego rezultatem zapytania /object[//object-[@class="a"] and //object[@class="b"]]. Zbiór ten można otrzymać za pomocą złączenia strukturalnego. Korzeń oraz wierzchołek *o* należą do poszukiwanego zbioru

Fig. 2 shows a portion of an example tree. Observe that all edges are dashed (i.e. those representing ancestor-descendant paths). There are two sets of nodes, labeled a and b, respectively. The membership in either set is defined by a value of a string attribute @class. We need to find the set of their common ancestors. First, let us notice that if auxiliary search indices on the @class attribute are available then sets a and b can be efficiently identified. We face now a more challenging task - for each pair of a and b members common nodes on the

path to the root are to be found. The relevant query in *XPath* is /object[//object[@class="a"] and //object[@class="b"]]. Our main objective in this paper is to optimize processing of such queries.

2. Hierarchical data and the implementation technique

In our model hierarchical data structure is implemented with labels. Labels make a special data type which enables the resolution of ancestor-descendant relation in an efficient manner. This is to say, with the label values available for two nodes it can be decided whether there is a path between these two nodes. The labels also order a tree according to the preorder enumeration (c.f. [3], [4], [5]). In our persistency scheme, the nodes are stored in a cluster index, ordered by preorder. (There are two classes of labeling: interval and prefix. The former defines relation by testing a containment of two intervals in a linearly ordered set:

$$a.left < d.left \wedge a.right > d.right > a \subset_D d$$

The prefix version checks whether one label is a prefix of another. To implement the preorder the prefix labels must also be linearly ordered.

In our version of structural join the differences between labeling types are transparent. In our implementation tree nodes expose abstract interface, which enables the resolution of two relations (preorder and ancestor). There are two functions: *anc* and *pred*, that encapsulate the actual labeling scheme. Evidently, the explicit structure of a tree can be easily recovered from a set of labeled data items. In fact, many of the native and hybrid XML databases maintain redundant data structures (both explicit and implicit) to accelerate certain types of queries.

3. Structural join

The essential breakthrough in the query processing was achieved in the fundamental work by Jagadish et al. ([6], [7], [8]), who introduced the method of structural join. This join is an operation on two data sets that produces another data set consisting of pairs of data items from two input sets. The nodes (data items) in each pair are coupled by either parent-child or ancestor-descendant relation. As any query can be decomposed into a set of primitive structural joins, the conventional processing of a query is based on materialized intermediate result sets. We call this the off-line method, because each step requires all input data should be available at the moment the join is evaluated.

On the contrary, deferred evaluation feeds the data as the iteration progresses. We follow [9] (D.E. Knuth) in our terminology, which makes a difference between on-line and off-line

algorithm. The deferred processing of structural join belongs to the first category as it feeds data as late as possible (i.e. the cursor on the data table is advanced only to the nearest row containing data for the next result tuple). Unlike the on-line algorithm its off-line counterpart reads data in advance. An analysis of the on-line processing advantages, particularly in the context of querying, is given in Section 3. Let us only note that the most evident feature of the on-line is that no intermediate result sets need to be generated and stored, because the result is immediately transferred to the client application. Let us keep in mind that an option to generate such a result is still available provided it is still advantageous for some other reasons.

```
public static void Stack_Tree_Desc(IEnumerable<Node> a,
                                   IEnumerable<Node> d)
{
  IEnumerator<Node> ia = a.GetEnumerator(),id=d.GetEnumerator();
  ia.MoveNext(); id.MoveNext();
  while (ia.Current!=eof || id.Current!=eof || stack.Count>0)
     if (ia.Current.StartPos>stack.Peek().EndPos &&
         id.Current.StartPos>stack.Peek().EndPos)
     {
      stack.Pop();}
     else if (ia.Current.StartPos<id.Current.StartPos)
      {
         stack.Push(ia.Current);
         ia.MoveNext();
       }
    else {
      foreach (Node n in stack)
      if (n != bos)
        rs.Append( new Node(n.beginPos,id.Current.beginPos));
        id.MoveNext();
     }
     rs.Append(eof);
```

Fig. 3. The original Stack-Tree-Descendant algorithm in the C# notation. The off-line version always generates a materialized set, which is subsequently either processed by other query evaluation primitives or submitted to the client

Rys. 3. Implementacja algorytmu Stack-Tree w C#. Wersja off-line generuje zmaterializowany zbiór wynikowy. Może on być następnie przekazany klientowi lub poddany dalszemu przetwarzaniu przez procesor zapytań

In [8] a special category called pipelined algorithm is identified. Pipelined execution resembles to some extend the on-line one but there is a subtle difference. The *Stack-Tree-Anc* [6] may need to generate result tuples only to provide the desired ordering of the output. The property of being pipelined enables only joins to advance in parallel as soon as partial result are available this however does not ensure that the internal processing of a structural join does not buffer intermediate results. The pipelined execution may be considered on-line as long as only the processing at the level above the single structural join is taken into account.

4. Deferred query (after D. Box and A. Hejlsberg [10])

First, let us note that deferred processing and the on-line one are interrelated. A deferred operator evaluating the result set does not evaluate the next tuple until it has been requested. Therefore, source data are read and resources allocation proceeds as late as possible and the release thereof as early as possible. Our method makes extensive use of the constructs available in the latest release of .NET framework which include *yield return* operator along with accompanying facilities available in LINQ. We are going to discuss this point in a few examples.

```
// declare a variable containing some strings
string[] names = { "Allen", "Arthur", "Bennett" };

// declare a variable that represents a query
IEnumerable<string> ayes = names.Where(s => s[0] == 'A');

// evaluate the query
foreach (string item in ayes)
  Console.WriteLine(item);

// modify the original information source
names[0] = "Bob";

// evaluate the query again, this time no "Allen"
foreach (string item in ayes)
    Console.WriteLine(item);
```

Fig. 4. Deferred query in LINQ. The *ayes* collection members are evaluated dynamically as *foreach* loop iterates. No materialized result set is generated. The result set reflects the most recent state of the underlying data source

Rys. 4. Opóźnione przetwarzanie w LINQ. Kolekcja *ayes* jest wyznaczana dynamicznie wraz z iteracjami pętli *foreach*. Zbiór wynikowy nigdy nie powstaje. Wartości reprezentują zawsze bieżący stan źródła danych

By the example in Fig. 4., we are able to tell the difference between on-line and off-line methods. If at the moment the *iterator* is set up (i.e. assignment to variable *ayes*) a materialized result set had been created then both *foreach* loops would have output the same values to the screen ("Allen", "Arthur"). However, the deferred *iterator* always accesses the current state. Thus, the second loop is going to output only "Arthur".

The crucial advantage of deferred version is better flexibility in representing the current state. At any moment, independently of the actual state, the operator is able to yield the current tuple and transfer the control to the client. Fig. 5 shows the simplest possible case of the use of the *yield return* instruction. The *intSeq* collection contains an infinite sequence of integers. The new members appear at the moment the caller iterates over the collection. There are no constraints whatsoever on the source code and this construct can be also used for the implementation of database join algorithms. Moreover, such a join algorithm is able to access nested cursors/iterators on its own. In this way arbitrarily complex queries can be evaluated

in a deferred mode. We are going to get back to this point while discussing the composition of operators.

```
public static IEnumerable<Uint32> intSeq()
{
   uint i=0;
        while (true) {i++;yield return i;if (i==Uint32.MaxValue)
        i=0;}
}
```

Fig. 5. An infinite sequence of unsigned integers implemented with the yield construct. The iteration resets the state to 0 when maximum value is reached

Rys. 5. Konstrukcja *yield* umożliwia tworzenie zbiorów o nieskończonej liczbie elementów. Osiągnięcie wartości maksymalnej zeruje aktualny stan

With the deferred operator, there is no need to fetch the source data in advance. Source data is fed in as the iteration over the result set progresses (i.e. in the C#/.NET context *MoveNext()* method is invoked). Due to this, processing of data streams is also feasible. Deferred query operator is able to pick the data from a stream. In this case the sole responsibility of the query operator is to maintain the current state of computation. If data stream does not keep the pace with the query processing then the operator is suspended until the data is available. Let us point out here that this ability is unavailable for hierarchical data, because inputs for structural join are to be sorted by preorder. Owing to the storage technique in cluster indices, the input data is always guaranteed to be sorted this way.

5. Deferred evaluation of structural join (the main result)

Our deferred structural join operator yields only one result tuple at a time. Depending on the context either the ancestor node or the descendant node is returned. When the node is requested as a potential member of a result set then the descendant node is pertinent. In the case of an embedded sub-query, when the external operator matches it against the current node, it expects to obtain a node on the ancestor side of the join. Observe, that the level of embedding can be arbitrary (e.g. *XPath* language does not impose any restriction on query embedding, much like SQL), and the query engine verifies only whether there exists non-empty result set matching the context node (c.f. our example in Fig. 2). The deferred structural join operator in C# notation is given in Fig. 4. This is only one particular instance (most complex however), joining along ancestor-descendant axis and returning the descendant side of the join. The fully fledged implementation of a query engine would offer multiple variants of the join operator varying with descendant/child join type and expected result set.

```
public static IEnumerable<Node> JD(IEnumerable<Node> a, IEnumerable<Node> d)
{
  Stack<Node> stack = new Stack<Node>();
  stack.Push(bos);
  IEnumerator<Node> ia = a.GetEnumerator(),id=d.GetEnumerator();
  ia.MoveNext(); id.MoveNext();
  while (ia.Current!=eof || id.Current!=eof)
  if (ia.Current.pred(id.Current))
  {
      while(!stack.Peek().anc(ia.Current)) stack.Pop();
      if (ia.Current.anc(id.Current)) stack.Push(ia.Current);
      ia.MoveNext();
   } else
   {

      while(!stack.Peek().anc(id.Current)) stack.Pop();
      foreach (Node n in stack)
      if (n != bos)
        yield return new Node(n.beginPos,id.Current.beginPos);
        id.MoveNext();
    }
      yield return eof;
  }
```

Fig. 6. Deferred structural join operator based on the yield-return construct, the ancestor/descendant version

Rys. 6. Opóźniony operator złączenia strukturalnego korzystający z konstrukcji yield-return. Wersja dla relacji przodek-potomek

As we can see the structural join is expected only that its two arguments implement the *IEnumerable* interface. Further, we see will that there is an additional restriction on the second argument as both inputs must be sorted according to pre-order. However, we will also see that this additional restriction does not compromise the flexibility and the expressiveness of our method. The off-line version can be easily obtained by simply replacing the two yield instruction with the respective appends.

As already mentioned in the introduction, we have slightly modified the original algorithm (Fig. 3), not only in terms of underlying labeling, but also the order of condition verification is different. Our version produces the same output although the exact number of comparisons is smaller. This, however, has rather negligible impact on the performance, which depends heavily on data access and I/O operations. A comprehensive model of the query algorithm performance given in [8] is based on input sizes, data access costs, I/O costs and stack operation with the stack costs probably being least significant.

The key performance factor is the fact, that each of the input data cursor needs to be scanned only once and no intermediate result is generated. When this feature is considered in the context of the structural join order optimization it becomes evident that on-line processing has the optimum execution time with respect to the input data size. We are also going to give some technique enabling a considerable reduction of the size of input data.

6. Composition of operators

The structural join is considered to be a certain primitive operation on hierarchical data [6]. To make our result complete we are going to show a method of operator composition, which accommodates arbitrary queries. Observe, that in the conventional method (i.e. off-line) such composition is not needed because it relies on intermediate result sets. An off-line kind (of query engine always processes such materialized result sets, until the final result is obtained.

First let us note that on-line operators are regular objects (i.e. instances of classes) and can be created the usual way by invoking the constructors. The structural join constructor always takes up at least two parameters representing the input data sets. The only constraint on the input data is that it must be ordered on the join field. Consequently, only the first argument can be a result of another join, the right (i.e. descendant or child side) must be a cursor on a data table (e.g. a cursor on an auxiliary index). As data tables are stored in a cluster index ordered according to preorder enumeration the input is always guaranteed to follow this ordering. Let us also observe, that join operator produces always output ordered by descendant side. Thus both sides fulfill the constraint. Let us now consider a path query expressed in *XPath*:

```
/alpha/beta/gamma/delta/epsilon
```

The evaluation of this query includes 5 index searches on auxiliary tag indices and 4 structural parent-child joins. If there are additional search conditions, these can be processed sequentially during the search. The translation into a series of structural joins (and thus a composition of operators) is as follows:

```
IEnumerable pathQueryOp=
                        JC(
                            JC(
                                JC(
                                    JC(
                                        DT("alpha"),
                                        DT("beta")
                                    ),
                                DT("gamma")
                                ),
                            DT("delta")
                            ),
                        DT("epsilon")
                        );
```

In queries containing ancestor-descendant joins the JC constructor is to be replaced by its path counterpart, i.e. JD. For /alpha//beta query, the corresponding code would be:

```
IEnumerable pathQueryOp= JD(DT("alpha"), DT("beta"));
```

The two kinds of constructors can be mixed in the query operator, because the only type checking is performed on the *IEnumerable* interface. Thus constructions like: *JC(JD(... , ...))* are also allowed. Let us notice that the composition introduces very little syntax overhead as compared to the original *XPath* notation.

7. Conclusions

The question of optimal data manager characteristics has not been studied yet. Let us only highlight one crucial improvement that is possible at this level. Whenever a next tuple is to be fetched from an auxiliary index, the join operator can provide data manager with the useful information on the nearest available position. Evidently, available data managers do not take advantage of such a clue, because this would require that a specialized search function is available.

Our future work will include extensive performance tests. The optimal complexity does not always guarantee the optimal performance. As already mentioned, apart from the join algorithm itself, the key factor is the operational characteristics of the underlying data manager.

It remains to be shown that nested queries (e.g. common ancestor query given in Fig. 2) can be evaluated with a single scan of the data cursor. This proposition is true, however the proof thereof has not been given in this paper. We are going to fill this gap in our subsequent paper.

BIBLIOGRAPHY

1. Fomichev A., Grinev M., Kuznetsov S.D.: Sedna: A Native XML DBMS. In Wiedermann J., Tel G., Pokorný J., Bieliková M., & Stuller J. (eds) Theory and Practice of Computer Science, 32nd Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2006, January 21-27 2006, Merín, Czech Republic, Berlin: Springer, (LNCS), p. 272÷281.
2. Pleshachkov P., Chardin P., Kuznetsov S.D.: A DataGuide-Based Concurrency Control Protocol for Cooperation on XML Data. In Eder J., Haav H.-M., Kalja A., & Penjam J. (eds) Advances in Databases and Information Systems, 9th East European Conference, ADBIS 2005, September 12-15, Tallinn, Estonia, Berlin: Springer, (LNCS), p. 268÷282.
3. Tropashko V.: Nested Intervals for Tree Encoding in SQL, ACM SIGMOD Record, 34(2), p. 47÷52.

4. Wu X., Lee M.L., Hsu W.: A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April, 2004 Boston, MA, USA. IEEE Computer Society, p.66÷78.
5. O'Neil P., O'Neil E., Shankar P., Cseri I., Schaller G., Westbury N.: ORDPATHs: Insert-Friendly XML Node Labels, SIGMOD, Paris 2004, p. 903÷908.
6. Al-Khalifa S., Jagadish H.V., Koudas N., Patel J., Srivastava D., Wu Y.: Structural Joins: A Primitive for Efficient XML Query Pattern Matching. Proceedings of the 18th International Conference on Data Engineering, ICDE 2002, 26 February - 1 March 2002, San Jose, CA. IEEE Computer Society, p. 141÷152.
7. Jagadish H.V., Al-Khalifa S., Chapman A., Lakshmanan L.V.S, Nierman A., Paparizos S., Patel J.M., Srivastava D., Wiwatwattana N., Wu Y.: A Native XML Database, VLDB Journal 11(4) 2002, p. 274÷291.
8. Yuqing Wu, Jignesh M. Patel, H. V. Jagadish: Structural Join Order Selection for XML Query Optimization. In Umeshwar Dayal, Krithi Ramamritham, T. M. Vijayaraman (Eds.): Proceedings of the 19th International Conference on Data Engineering, ICDE 2003, March 5-8, 2003, Bangalore, India. IEEE Computer Society, p. 443÷454.
9. Knuth D.E.: The Art of Computer Programming. Vol. 2. Seminumerical algorithms. Addison-Wesley, 1969.
10. Box D., Hejlsberg A: LINQ: .NET Language-Integrated Query, MSDN, Microsoft Corp., February 2007.

Recenzent: Dr inż. Marcin Gorawski

Wpłynęło do Redakcji 20 stycznia 2009 r.

Omówienie

Efektywne przetwarzanie złączenia strukturalnego stanowi ważne zagadnienie w dziedzinie baz danych XML oraz efektywnego przeszukiwania i przetwarzania hierarchicznych struktur danych. W pracy [6] zamieszczono fundamentalny algorytm umożliwiający jednoprzebiegowe wyznaczenie złączenia dwóch podzbiorów wierzchołków drzewa. Złączenie takie odbywa się względem relacji przodek –potomek lub rodzic dziecko danych. W pracy podajemy zmodyfikowaną implementację oryginalnego algorytmu w języku C# (rys. 3).

Czynnikami decydującymi o efektywności przetwarzania zapytań są operacje I/O [8]. Liczba tych operacji zależy zarówno od rozmiaru danych wejściowych, jak i liczby oraz wiel-

kości pośrednich zbiorów wynikowych. Innym czynnikiem jest wybór porządku, w jaki generowany jest zbiór wynikowy – porządkowanie względem potomków jest szybsze niż względem przodków. Korzystne pod względem wydajności jest więc ograniczenie przetwarzania do pierwszego sposobu uporządkowania.

W pracy zaprezentowano podejście polegające na przetwarzaniu asynchronicznym. Zaproponowany operator opóźnionego przetwarzania (rys. 6) sprawia, że zbiory pośrednie nie są już potrzebne. Opis operatora wraz z kodem zawarty jest w p. 5. W implementacji posłużono się konstrukcjami języka C#. W punkcie 4 opisano ułatwienia dotyczące przetwarzania opóźnionego dostępnych w .NET od wersji 3.5. Należy zauważyć, że techniką tą można posługiwać się niezależnie od środowiska programowania. Brak konstrukcji ułatwiających (*yield-return*) powoduje jedynie, że zapis jest mniej czytelny.

Zaproponowany operator może być używany do przetwarzania dowolnie złożonych zapytań ścieżkowych (np. *XPath*). Jest to możliwe dzięki mechanizmowi kompozycji operatorów. Wywołanie konstruktora nie generuje zbioru wynikowego, a jedynie inicjuje odpowiednio stan przetwarzania, który następnie jest sukcesywnie aktualizowany w miarę pobierania wyników przez klienta. Zatem kolejność wywoływania konstruktorów nie ma nic wspólnego z późniejszym przetwarzaniem danych. Kompozycja operatorów dotyczy trzech klas obiektów implementujących wspólny interfejs: operatora w wersji przodek-potomek (JD), rodzic-dziecko (JC) oraz tablicy danych (DT).

Addresses

Andrzej SIKORSKI: Politechnika Poznańska, Instytut Automatyki i Inżynierii Informatycznej, ul. Piotrowo 3A, Poznań, Polska, andrzejs@et.put.poznan.pl.
Jarosław CIEMNOCZOŁOWSKI: ul. Wielichowska 5, 61-418 Poznań, Polska, j.ciemnoczolowski@pcnet.net.pl.