AUTOMATYKA/ AUTOMATICS • 2012 • Vol. 16 • No. 2

http://dx.doi.org/10.7494/automat.2012.16.2.165

Andrzej Sikorski*

# Fiber Processing of Queries on Hierarchical Data

## 1. Introduction

Our research is deals with the effective querying of XML documents. We employ fibers – execution units which support flexible control flow – to host tailored search steps with an intention to find a query evaluation plan that minimizes the most relevant performance measures. Fibers are cheaper than threads, store locally the current computation state and allow a scalable allocation of resources (i.e. execution is scheduled parallel on multi CPU/core machines but sequential otherwise). They are useful programming constructs for concurrent processes and streamlined implementation of entangled routines 0. The generic lifecycle of a fiber is given in Figure 1a. It can yield the control at any moment, keeping its current state. As soon as other component (possibly a fiber) issues a suitable signal it is possible to resume execution (movenext in Fig. 1a. – our notation is referring to LINQ).

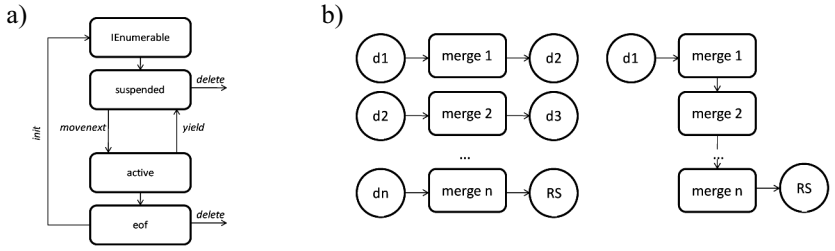


**Fig. 1.** The lifecycle of a LINQ fiber – c.f. [3] (a). Hosting each merge of a multiphase join algorithm in a dedicated fiber [2]removes intermediate files $d_i$ $2 \le i \le n$ (b)

The idea of using fiber processing for merge sort optimization was originally mentioned by D.E.Knuth in [2]. In general, multiple pass algorithms can be optimized if execution is distributed over fibers (c.f. the n pass merge in Fig. 1b.). In this way no intermediate sets are generated. In the case of database systems (in particular XPath evaluators operating on XML documents), excessively large intermediate result sets are liable to deteriorate the IO

* Technical University Poznań, Faculty of Electrical Engineering, Piotrowo 3A, 60-965 Poznań, Poland

performance. Merge sort is the main technique of determining relational join, for this reason fiber processing is so widespread in the SQL world.

The hierarchical counterpart of the relational join is the Structural Join we are concerned with. The Structural Join is a non-commutative set-valued operator of two arguments that are subsets of tree nodes. The value is another subset consisting of nodes of the right-side argument which are either descendants or children of the left-side one. Figure 2 gives an example. The problem statement is simple but the real challenge is doing this effectively on large data sets, that is why the Structural Join implementation is a topic of current interest in database research.

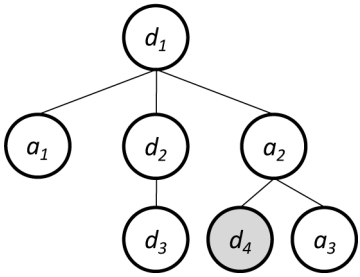


**Fig. 2.** The lifecycle of a LINQ fiber – c.f. [3] (a). Hosting each merge of a multiphase join algorithm in a dedicated fiber [2] removes intermediate files $d_i$ $2 \leq i \leq n$ (b)

The focus of our research pertains to optimization of the Structural Join (SJ) algorithm from [3] – namely the Merge Stack Join. We consider Merge Stack Join in the context of its interactions with a data manager. The data manager in use with us is taking care of tasks relevant to sequential scanning, index searches and concurrency. On the physical level data is stored in memory pages that make up a data processing unit for IO. The definitive factor for our considerations here is a property of the manager: if one data object is requested the entire page must be fetched into RAM. The logical organization of data is based upon a preorder cluster index where labeled (c.f. [4]) data nodes (or references thereto) are stored. We introduce a fiber SJ (i.e. deferred) operator that renders itself easily for various optimization techniques, taking advantage of input characteristics.

## 2. Related Work

Related work [4, 7] dealing with efficient processing of path queries introduced the use of cluster preorder index as a method for boosting the performance of SJ. O'Neil *et al.* ([4]) gave a comprehensive description of various evaluation scenarios in the context of XML facilities for MS SQL featuring proprietary ORDPATH's labeling system. In particular, the usefulness of various auxiliary indices which accelerate searches on large documents was discussed. In [7] apart from accounting for cluster index utilization there was a novelty indexing scheme that arranges identically tagged nodes in an ancestor-descendant relationship. This index, named C-Forest was aimed at speeding up the identification of the next candidate input node, likely to be a member of the result set.

Our work takes a different approach and is inspired by [5], which set forth an elaborate performance model of structural join processing. We decompose the query operator into individual execution fibers and apply optimization techniques locally, independently of each other. Our objective is to optimize the measures from [5], taking into account the scenarios from [4]. Intermediate sets were also removed in [6] where Holistic Twig Join (HTJ) was defined. What they did in this work was an approach consisting in processing a path/twig query in its entirety. In consequence, complex control flows among query primitives did not appear. Unlike [6][6] we still go on with the independent processing of individual joins, solving two issues typical for HTJ – sub-optimality of both parent-child joins and IO operations on inputs. An attempt to tackle the IO issue was made in [7]. However, we proceeded along different lines, optimizing each fiber hosted SJ individually while preserving the optimal performance with respect to both the measures from [5] and improving IO complexity.

## 3. Fiber Evaluation of Structural Join

Our fiber hosted operator locates only one result tuple at a time and does not generate an intermediate result set. Complex path queries can be obtained by a composition of multiple operators (i.e. aggregation of fibers) that embeds them recursively. The fiber SJ operator is given in Figure 3. This is the most general version, joining along the ancestor-descendant axis and returning to the descendant side of the join. The fully fledged implementation of a query engine would offer multiple variants of the join operator varying with a descendant/child join type and expected result set; modifications are straightforward. We are going to take a closer look only at those which offer optimization opportunities.

```
01 operator DqSJDesc(a,d)
02 while(¬eof(a) ∨ ¬eof(d))
03 if(a<d)                    //nodes compared
                               //according to preorder
04         while ¬(stack.top ⊃ a) pop(stack)
           //stack.top ⊃ a - a is contained in
           //stack.top subtree
05         movenext(a)
06 else
07         while ¬(stack.top ⊃ d ) pop(stack)
08         if (stack ⊃d) yield d
09         movenext(d)
   end if
10 yield eof
```

**Fig. 3.** Joining two input sets ($a_i$, $d_i$) of tree nodes with a structural join along child axis, $d_4$ is the only member of the resultset

Our fiber SJ algorithm differs significantly from the original one: appends are replaced with yield operations and, in consequence, no result tuples are constructed. The current state is represented with only one side of the join and persistent ancestor-descendant pairs are not needed. When a candidate tuple is located, a fiber yields control to the caller, which is, depends on the logical structure of the query, either the final information consumer or just another fiber. In the case of deep-left composition of operators (c.f. [9]) after a matching descendant is located the ancestor side is no longer needed. This also drops the necessity of the more expensive ancestor ordered version of SJ (c.f. [3] on the cost of the ancestor). Unlike the original SJ there is also no need to iterate the ancestors stack. Much like in the case of HTJ we assume the current path has already been determined and the matching condition for the descendant is verified. A full reference on these issues can be found in [3].

Our order of condition verification is different, but functionally equivalent to the original, as the output is the same although the exact number of comparisons is smaller. This, however, has a fairly negligible impact on performance that depends heavily on data access and IO operations. The performance model given in [5] is based on input sizes, data access costs, IO costs and stack operation with the stack costs being least significant. The key performance factor is 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 deferred processing has the optimum execution time with respect to the input data size. In the next section we are going to give some technique enabling a considerable reduction of the size of input data, introducing tailored search steps.

## 4. Tailored Search Steps

Contrary to HTJ the deferred operators evaluating individual joins allow for a more flexible computational strategy. Keep in mind, that [7] implies restriction pertaining to HTJ optimality in the case of parent-child join. An *XPath* query can consist of multiple search steps, with varying characteristics. Thus, we introduce two optimizing options – employing FIFO (level, preorder) index for parent-child and an overloaded descendant list iterator (i.e. *movenext* with an additional hint). Fibers hosting tailored and general operators are compatible with each other and can be freely aggregated in each other to make up path query operators.

### 4.1. Employing FIFO Index

Joining a node with its children is more selective than the same with descendants – a lot can be gained if children are processed by the FIFO order. Consider the join in Figure 4 – the current operator position is the $(a_i, d_j)$ tuple and the next child $d_{j+k}$ is located far, possibly in another memory page. Standard processing based upon a preorder cluster would unnecessarily scan $k$–1 nodes coming in between two siblings.

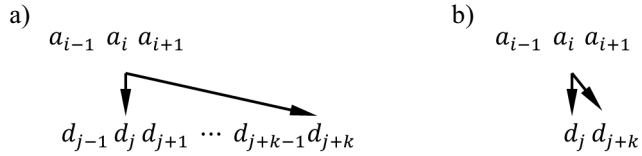


**Fig. 4.** Two siblings labeled $d_j$ and $d_{j+k}$ are separated by *k–1* descendant nodes (a). When the descendant side is processed in FIFO order $d_{j+k}$ is a direct successor of $d_j$ (b)

The FIFO order makes $d_{j+k}$ a direct successor of $d_j$. Due to this, we can save on a significant number of IO accesses, otherwise necessary to scan the sub-tree rooted at $a_i$. Now, it is sufficient to note, that the tailored operator using FIFO does not alter the output ordering. Therefore, optimization is transparent for other query components. The modifications of the general deferred operator are given in Figure 5. After the ancestor side has been advanced, the operator locates the first child (line 05), calling the appropriate variant of *movenext*. If there is no such node, *d* is positioned in such a way that it would force a subsequent shift of the ancestor (in our production version we simply implemented an embedded loop which advances the ancestor and updates the stack as long as no matching child exists). Observe that the second argument in the descendant branch (line 09) is used merely to resolve correctly among multiple overloaded operators. This variant of *movenext* function advances the cursor sequentially along the FIFO index – the children are located in consecutive index entries.

```
 …
04    while ¬(stack.top ⊃ a) pop(stack)
05    movenext(a)
05a   movenext(a,d)//FIFO locates the first child
 …
09    movenext(d,FIFO)//advances  FIFO
```

**Fig. 5.** Modifications of DqSJDesc operator for tailored parent-child search.

## 4.2. Additional Hint for the Cursor Shift

For highly selective ancestor-descendant join we may use a tailored shift function, iterating the descendant input list. This is happening with an ancestor list of much lesser cardinality than the descendant. Consider Figure 6 – if the gap between two subsequent ancestors exceeds the size memory page, some IO operations on the descendant side may be omitted.

The upper index of the descendant node denotes the memory page ID. If $a_i << a_{i+1}$ then it is possible to skip memory pages numbered from *j* to *l*. We are getting this by replacing sequential scanning with index positioning. For this purpose we must implement another overloaded *movenext* variant, that takes current ancestor node as a parameter. If an inspection of label values for two consecutive ancestors suggests that the descendant resides on a distant page, a number of IO accesses is skipped (in Fig. 6 the *k* page would not be fetched).

Moreover, most likely $a_{i+1}$ will be co-located with its first descendant on the same page, which may result in an additional performance gain.

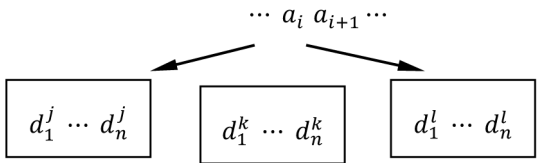


**Fig. 6.** A large gap between two consecutive ancestors hints that memory pages can be skipped

```
04a  a1←a
05   movenext(a)
…
08   if (stack ⊃ d) yield d
08a  if dist(a1,a)>threshold
      movenext(a,d,PREORDER) else
09   movenext(d)
```

**Fig. 7.** The function dist calculates distance between two labels, the threshold is a function of the page size

## 5. Composition of Operators

The structural join is considered to be a certain primitive operation on hierarchical data [3]. To make our result complete we are going to show a method of operator composition, which accommodates path queries. Observe, that in the conventional method (i.e. off-line) such a 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 fibers are regular objects (i.e. instances of classes) and can be created in the usual way by invoking the constructors with parameters. The structural join constructor always takes up at least two such 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 the input is always guaranteed to follow preorder. Let us also observe that join operator produces always an output ordered by the 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 sequence of SJs (and thus a composition of operators) is the following (C#):

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

In queries containing ancestor-descendant joins the *edge* 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”));
```

Various kinds of constructors can be combined in the query operator, because the only type checking is performed on the *IEnumerable* interface. Thus, constructions like: JC(JD( ... , ... )) are also allowed.

## 6. Experimental Evaluation

Our experimental test-bed consists of two components: data manager and query evaluator. We made use of Berkeley DB, a small footprint transactional database library, supporting data persistency and indexing. We implemented the query evaluator as a native Win32 application in C++, compiled with MS Visual Studio 2010. The testbed was deployed on a 2.2 GHz PC with 2 GB RAM and Windows Vista operating system.

The performance evaluation was based upon a method from 0 with queries run on a large (123MB), recursive “organization” document (manager-department-employee-email). Our objective was merely confined to evaluate the fiber SJs, therefore a complete XQuery processor was not necessary. All queries were instances of a regular path pattern of the form MxDyE, like:

E → //Employee,
M2E → //Manager//Manager//Employee,
M1D2E → //Manager//Department//Department//Employee.

An instance of XML DB Sedna [10], deployed on an identically configured PC, was used for comparison.

Table 1 contains data about the MxD3E query execution time and result set cardinality (x ranges from 3 to 1, DQE – fiber SJ, DG – DataGuide Sedna). Figure 8 shows a comparison of the result set cardinality and the execution time of the MxE queries (x ranges from 6 to 1) and Figure 9 confronts the performance of fiber SJ (DQE) with Sedna (DG).

**Table 1**
Manager-Department (MxDyE) queries performance,
fiber DQE vs. DG (DataGuide –Sedna) Processing time in seconds

| | DQE | DG | card. |
|---|---|---|---|
| M3D3E | 2.085 | 4.854 | 558074 |
| M2D3E | 2.253 | 6.986 | 790953 |
| M1D3E | 2.542 | 17.312 | 1428389 |

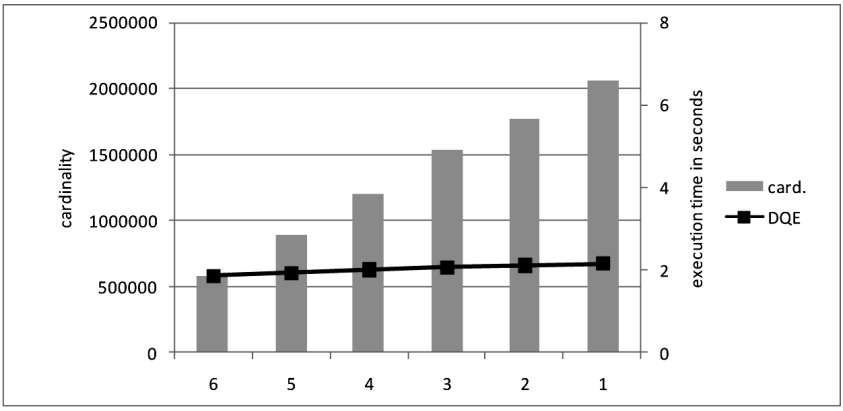


**Fig. 8.** The performance of MxE queries (DQE) vs. result set cardinality.
The cardinality has only a minor impact on the performance

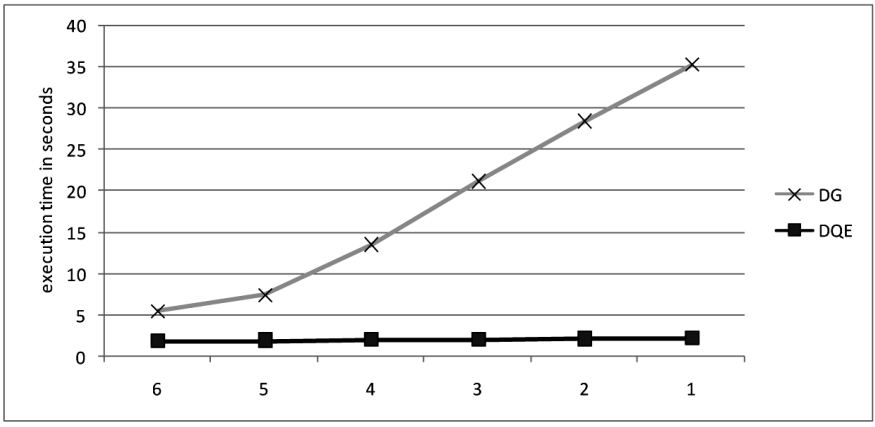


**Fig. 9.** The performance of MxE queries, fiber DQE vs. DG. Processing time in seconds.
The fiber based join significantly outperforms the DG

## 7. Conclusions and Subsequent Work

In subsequent work we will provide a proof of conditional optimality with respect to IO complexity. Apart from this we will deal with an extension of the path composition of SJs into a twig one. The twig composition is surprisingly straightforward and SJ deferred operators to be compliant do not require any modification. The optimality of the technique given in Sec. 4.2 appears to be confined to selective SJs, i.e. the path pattern has to pick a relatively small result set from the input.

### References

[1] Shankar A., *Implementing Coroutines for .NET by Wrapping the Unmanaged Fiber API.* MSDN Magazine, Microsoft, Redmond, 2003.

[2] Knuth D.E., *The Art of Computer Programming. Vol. 1*. Fundamental Algorithms. Addison-Wesley, Reading, 196, 248.

[3] Al-Khalifa S., Jagadish H.V., Koudas N., Patel J., Srivastava D., Wu Y., *Structural Joins: A Primitive for Efficient XML Query Pattern Matching.* In: Proc. of the 18th International Conference on Data Engineering. IEEE Computer Society, San Jose CA, 2002, 141–152.

[4] O'Neil P., O'Neil E., Shankar P., Cseri I., Schaller G., Westbury N., *ORDPATHs: Insert-Friendly XML Node Labels, SIGMOD*, Paris, 2004, 903–908.

[5] Wu Y., Jignesh M., Patel H., Jagadish V., *Structural Join Order Selection for XML Query Optimization*. In: Dayal U., Krithi Ramamritham K., Vijayaraman T.M. (eds.): Proc. of the 19th International Conference on Data Engineering, IEEE Computer Society, Bangalore, 2003, 443–454.

[6] Bruno N., Koudas N., Srivastava D., *Holistic twig joins: optimal XML pattern matching*. SIGMOD Conference, ACM, Madison, 2002, 310–321.

[7] Fontoura M., Josifovski V., Shekita E.J., Yang B., *Optimizing cursor movement in holistic twig joins*. CIKM, ACM, Bremen, 2005, 784–791.

[8] Box D., Hejlsberg A.: LINQ, *NET Language-Integrated Query*. MSDN, Microsoft Corp., Redmond, 2007.

[9] Graefe G., *Query evaluation techniques for large databases*. ACM Computing Surveys, 25(2), 1993, 73–170.

[10] 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, LNCS, vol. 3831, Springer, Heidelberg, 2006, 272–281.