Proceedings of the 7th International Conference on Applied Informatics
Eger, Hungary, January 28–31, 2007. Vol. 2. pp. 181–188.

Parallel processing search engine for very large XML data sets*

Vu Le Anh, Attila Kiss, Zoltan Vinceller

Department of Information Systems, ELTE university, Hungary
e-mail: leanhvu@inf.elte.hu, kiss@ullman.inf.elte.hu, vzoli@inf.elte.hu

Abstract

In this paper, we study several fundamental problems for building a parallel processing search engine for very large XML data sets. In our model, the data set can be considered as a very large XML file, which is fragmented and stored on different machines. The machines are connected by the high speed local network and the query language is based on regular queries. The following problems are introduced and studied: 1. The general model for the system; 2. The query language; 3. The efficient query processing algorithm; 4. The efficient fragmentation algorithm. All introduced algorithms and solutions bring into play the extensibility and the self-describing of XML data sets, and they also promote the abilities of the parallel systems.

Keywords: Search Engine, Parallel Processing, Regular Queries, XML

MSC: 68P05, 68P20

1. Introduction

Nowadays highly parallel database systems displace traditional mainframe computers for the largest database and transaction processing tasks. The architecture of these systems is based on a shared-nothing hardware design [1] in which processors communicate with one another only by sending messages via a high speed local network. The data set is fragmented across disk storage units attached directly to each processor which allows multiple processors to scan the large data set in parallel. These systems are easy to be expanded and powered by adding new machines, new RAM memories and disks, which become more and more cheaper and faster nowadays. On the other hand, XML has become the dominant standard for exchanging and querying documents over the Internet. XML offers its users many advantages, especially in self-describing, extendibility and inter-operability.

*The authors thank the (partial) support of the Hungarian National Office for Research and Technology under grant no.: RET14/2005.

XML is a good choice for representing various types, dynamic schema or open data sets such as scientific databases, clustered web storages, open directories, etc.

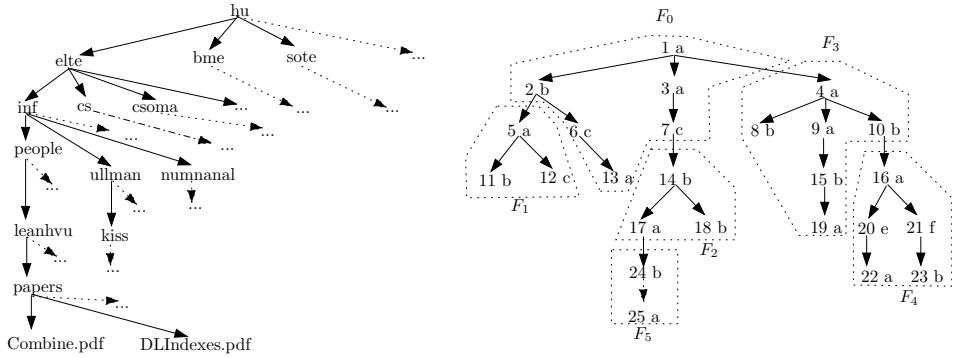
In this paper, we study several fundamental problems for building a parallel processing search engine for a very large XML data set. Firstly, we study the model for our system. The XML data set is a labelled tree. The data tree is fragmented into a collection of disjoints subtrees (or *fragments*) storing on the machines (or *sites*). The number of fragments is considered constant. There is no replication in our context. Fragments can be split, unified or changed from some site to the other. The model is dynamic, adaptive and brings into play all the advantages of XML and parallel database systems. Secondly, we introduce a simple query language for our system. Our querying language is a class of regular path expressions, which are the core of the most popular querying languages proposed for XML such as XPath [4], XQuery [5], XML-QL [6]. The query language supports string pattern so that retrieves label values more powerful. Thirdly, we study an efficient regular query processing algorithm based on the partial evaluation [2, 3]. The partial evaluation supports parallelism and guarantees the communication cost depends only on the result of the query. However these algorithms have redundant operations. In our algorithm, the redundant operations are rejected by preprocessing over the tree index. The size of the tree index is considered as constant and the cost of preprocessing is ignored. Our algorithm is more efficient than the older versions. Finally, we study an on-the-fly fragmentation algorithm, which holds the balance of the works between sites. We introduce a simple statistical system for measuring the cost over each site and fragment. The statistical system and the union-, split- and change fragment operations help us to manage the fragmentation efficiently.

The organization of our paper is as follows. We study the model of the system in Section 2. We introduce a simple query language for our systems in Section 3. The efficient query processing algorithm is introduced and studied in Section 4. The fragmentation algorithm is introduced and studied in Section 5. Section 6 concludes our paper.

2. The model of the system

Let us see the example of very large XML shown in Figure 1(a). We represent all public files of hungarian universities on Web by a XML tree. Each public file is stored in each leaf of the tree. The path from the the root to the leaf is determined by the URL address of this file. The absolute path of the leaf containing file `http://people.inf.elte.hu/leanhvu/papers/Conbined.pdf` is `/hu/elte/inf/people/leanhvu/papers/Combined.pdf`. All the public files of the ELTE university are stored in the leaves under the node `/hu/elte/`. Clearly, with each node is labelled, the tree XML is self-describing but has no fix schema. The users can retrieve all information of the nodes. In the case the tree is big, it is fragmented and stored in different sites.

Formally, the data set is modeled as a *labeled tree* T. Each node v of T is labeled by a label value $label(v) \in \Sigma$ (Σ is an alphabet). T is fragmented by a collection



(a) XML tree for public files of Hungarian universities on Web (b) Fragmented XML Tree T

Figure 1: Example of fragmented XML trees

F of disjoint trees, or *fragments* F_i. The fragments are stored in different *sites*, S_j. The set of sites are denoted by **S**. The sites are connected with each others by high speed interconnection. The numbers of sites and fragments are considered as constants. The fragment containing the root of the tree T, is called the *root fragment* denoted by F_{root}. In Figure 1(b), this is fragment F_0. F' is the *parent-fragment* of F if there exists node $v \in F'$ such that the root w of F is a child of v in the original tree T. In Figure 1(b), F_2 is a parent-fragment of F_5.

3. The query language

Our query language is a class of regular path expression. Here is the grammar:

$$Exp \leftarrow \quad Exp|Exp \;\mid\; /P \;\mid\; //P$$
$$P \leftarrow \quad \epsilon \;\mid\; < StringLiteral > \;\mid\; P/P \;\mid\; P//P$$

ϵ is the empty word. $< StringLiteral >$ can be a value of Σ or a subset of Σ matching a string pattern. /, // and | signs are borrowed from XPath [4], and they have the same meaning as in XPath. Here are several examples of our queries over the XML tree in Figure 1(a):

1. `//leanhvu/papers/-`: Finding all papers of `leanhvu`.
2. `//leanhvu/papers/%.pdf`: Finding all papers of `leanhvu` in PDF file format.
3. `//leanhvu/papers/-~|~//kiss/papers/-`: Finding all papers of `leanhvu` or `kiss`.
4. `/hu/elte/inf//%anal%`: Finding all documents or directories under the domain `inf.elte.hu`, whose name contains "anal" substring.

In these examples, `-, %.pdf, %anal%` are string patterns and `hu`, `kiss`, `leanhvu`, `papers`, `elte`, `inf` are label values.

A data node *matches the query* if the label path from the root to the node matching the query. The *result of the query* on the XML tree is the set of data nodes

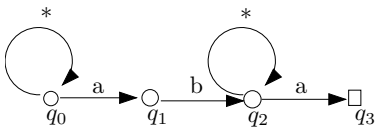


Figure 2: The query graph of the query $Q = //a/b//a$

matching the query. Each query is a regular expression and can be represented by an automata. The graph of the corresponding automata is called *query graph*. The result of the query on the XML tree can be determined by parsing the query graph and the data graph with following rules: (1) beginning at (u_0, q_0) (u_0, q_0 are roots of T and Q respectively); (2) From (u, q) we visit (v, p) for each v is a child of u and there exists a transition from state q to state p with the label of u. (3) u is an element of the result if some (u, q) is visited and there exists a transition from state q to final state p with the label of u. Each pair (u, q) is visited maximum one time as the XML tree is acyclic. An example of a query and the corresponding query graph is shown in Figure 2. The result of this query over the tree data in Figure 1(b) is $\{5, 13, 16, 19, 22, 25\}$.

4. The query processing algorithm

There are two criteria for the effectiveness of a query processing algorithm in the parallel database systems. The first criterion is the *responding time* for the answer. The second criterion is the *total cost* of processing and communication of all sites. The responding time could be minimized by parallelism. The total cost must be reduced in case we serve many users in the same time. As we have known, there are two approaches for processing regular queries: *stream processing* and *partial parallel processing*. There is no parallelism in the stream processing. With the partial parallel processing, each site can compute the partial result in parallel. However in the older versions [2, 3] of the partial parallel processing for regular queries have redundant operations which can make the total cost be high. Because the limit of the paper, we just describe our algorithm as an improved version of partial parallel processing for regular queries with no comparison with the other approach or the older versions.

The FRAGMENT-PROCESS(F,q) operation returns the subset of the result in F when we parse F and the query at the root of F and state q. The MATCHING(q,*label*) function returns the set of transforming states q' when the corresponding non-determined automata stands at state q and receives the sign *label*. The correctness of the operation can be examined easily by the rules in section 3 for computing the result. Let us see the fragmented tree shown in Figure 1(b) and the query shown in Figure 2. The operation (F_4, q_2) returns {16,22} and (F_2, q_0) returns empty set.

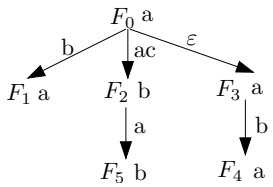


Figure 3: The index of the XML tree T in Figure 1(b)

FRAGMENT-PROCESS(F,q)
begin
1. $FragResult \leftarrow \emptyset$
2. SCAN($root(F)$,q,F)
3. **return** $FragResult$
end

MATCHING(q,$label$)
begin
1. $S \leftarrow \emptyset$
2. **for each** transition $(q, label, p)$ **do**
3. $S \leftarrow S \cup \{p\}$
4. **return** S
end

SCAN(u,q,F)
begin
1. **for each** state $p \in$ MATCHING($q, label(u)$) **do**
2. **if** p *is final state* **then**
3. $FragResult = FragResult \cup \{u\}$
4. **for each** edge (u, v) **and** $v \in F$ **do**
5. SCAN(v,p,F)
end

In the older versions of the partial parallel processing approach, all possible Fragment-Process operations are executed as they assume that they just know about the information of links between fragments and no more. Clearly, the operation (F, q) is *redundant* or *unreachable* if $(root(F), q)$ is not visited from the rules in Section 3. In above example, in the case fragment (F_2, q_1), (F_2, q_2) and (F_2, q_3) are redundant operations. We determine the *reachable* operations by preprocessing over the *tree index*, T_I, which is defined as follows:

Definition 4.1. Tree $T_I = (V_I, E_I)$ is the tree index of the XML tree T fragmented by **F**, in which: (i) $V_I =$**F** and each index node $F \in V_I$ is labeled by the root of fragment F and (ii) (F, F') is an index edge if F is parent fragment of F'. (F, F') is labeled by the label path of path p, which is the path connecting from the root of F to the root of F' but we reject two roots.

In the PROCESS-INDEX(T_I,Q) function, RO containing the list of reachable operations. The first reachable operation is $(F_{root}, Root(Q))$, where $Root(Q)$ is the start state of the automata. The MATCHING-II((F, F'),q) returns the set of states q' such that if the operation (F, q) is reachable and F is the parent-fragment of F' then the operation (F', q) is also reachable. The reachable operations in our example are (F_0, q_0), (F_1, q_0), (F_1, q_2), (F_2, q_0), (F_5, q_0), (F_5, q_1), (F_3, q_0), (F_3, q_1), (F_4, q_0), (F_4, q_3).

```
PROCESS-INDEX(T_I,Q)
begin
1. RO ← ∅
2. Stack ← ∅
3. Stack.push((F_root, Root(Q)))
4. while Stack ≠ ∅ do
5.   (F, q) ← Stack.pop()
6.   RO ← RO ∪ (F, q)
7.   for each index edge (F, F') do
8.     for each q' ∈MATCHING-II((F, F'), q) do
9.       Stack.push((F', q'))
10.return RO
end
```

```
MATCHING-II((F, F'),q)
begin
1. S ← MATCHING(q, label(Root(F)))
2. Let ⟨l_1, ..., l_k⟩ be the label of (F, F')
3. for i = 1 to k do
4.   S' ← ∅
5.   for each state q ∈ S do
6.     S' ← S'∪ MATCHING(q,l_i)
7.   S ← S'
8. return S
end
```

We choose the *master site* from the sites to control the process. The tree index is stored in the master site. The steps of the query processing algorithm are as follows:

1. *Determining reachable operations at master site.*
2. *Sending the reachable operations to each site.*
3. *Computing the reachable operations at each site in parallel.*
4. *Sending the results from some site to the master site.*
5. *The master waits until receiving all site results and determining the whole result.*

Clearly, the number of communications between sites is constant. The communication cost depends only the result of the query. The cost of computing reachable operations can be considered constant as the number of fragments and the size of the tree index are considered constants

5. The Fragmentation Algorithm

Statistical system. Because the cost of preprocessing over the tree index is ignored, the cost of processing and communication of the system is considered as the total cost processing in each fragment and sending the results to the master. Let $t_{F,Q}$ be the cost of processing (executing the FRAGMENT-PROCESS operations) and sending the results of these operations to the master site when processing query Q over fragment F. Let $\mathcal{Q}$ be the list of processed queries sorting by time in duration time D. The total cost over fragment F in a duration D is $t_{F,D} = \sum_{Q\in\mathcal{Q}} t_{F,Q}$. The total cost over site S in duration D is $t_{S,D} = \sum_{F\in\mathcal{S}} t_{F,D}$. The total cost of the system in duration D is $t_D = \sum_{S\in\mathbf{S}} t_{S,D}$. The number of data nodes of each fragment F is $nodenum(F)$.

Fragment operations. The ADD-FRAGMENT(S, F) procedure will add fragment F to site S. To avoid the increase of the number of the fragments we will unify the fragments in the same site into the only fragment in following cases: (i) F' is the parent-fragment of F'', F' and F'' are unified naturally as F (see Figure 3(a)) (ii) the roots of $F_1, F_2, \ldots, F_n$ is the same node stored in another site (see Figure 3(b)).

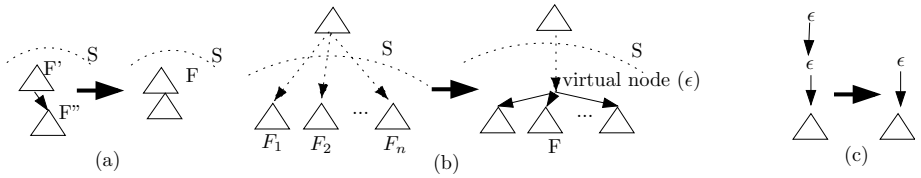


Figure 4: Unifying the fragments

They are unified by adding a virtual node as the root of the union-fragment and linking this virtual node to the roots of children-fragments. The virtual node is labelled by ϵ so that it will be ignored when processing the queries. The virtual nodes are also reduced by the rule shown in Figure 3(c) so that they must be the roots of the the fragments. The $F' = \text{SPLIT-FRAGMENT}(F, n)$ function will split F fragment into two parts in which the first part F' is the biggest subtree of F whose the number of nodes is smaller than n. In the case unifying the fragments, the cost over the union-fragment t_{F_u} is the total of the cost of the component fragments. In the case spiting fragments, the cost over the split-fragment $t_{F_{split}} = \frac{t_{F_{old}}.nodenum(F_{split})}{nodenum(F_{old})}$. The REMOVE-FRAGMENT$(S, d)$ function returns a fragment of S, whose the cost is smaller than d, and we remove this fragment from S.

Fragmentation Algorithm. ϵ, $\mathcal{T}$ are two chosen parameters ($0 < \epsilon < 1$, $\mathcal{T} > 0$). Our on-the-fly fragmentation algorithm manages the fragmentation satisfying following condition: *If* $t_D > \mathcal{T}$ *then* $\forall\ S_i, S_j$, $t_{D,S_i} \leqslant t_{D,S_j}$ *and* $t_{D,S_j} > 0$, *we have* $1 \geqslant \frac{t_{D,S_i}}{t_{D,S_j}} \geqslant 1 - \epsilon$. The $t_D > \mathcal{T}$ condition guarantees that the duration must be enough long, the second condition implies that there is no big difference between the works of the sites. The long duration guarantees the accuracy of the statistical system and the stability of the fragmentation.

Let D be a duration longer than $\mathcal{T}$; $t_{max} = max\{t_{S,D}\}$, $t_{min} = min\{t_{S,D}\}$, $\bar{t} = avg(\{t_{S,D}\})$ and $d = \epsilon.\bar{t}$. In the case $t_{max} - t_{min} \leqslant d$, we have: $\forall\ S_i, S_j$, $t_{D,S_i} \leqslant t_{D,S_j}$ and $t_{D,S_j} > 0$: $1 \geqslant \frac{t_{D,S_i}}{t_{D,S_j}} \geqslant \frac{t_{min}}{t_{max}} = 1 - \frac{t_{max} - t_{min}}{t_{max}} \geqslant 1 - \frac{\epsilon.\bar{t}}{t_{max}} \geqslant 1 - \epsilon$. Clearly if $t_{max} - t_{min} > d$, the fragmentation is not "good". Our fragmentation algorithm is as follows:

FRAGMENTATION$(\mathbf{F}, \mathbf{S}, D, \mathcal{T}, \epsilon)$
begin
1.**if** $t_D > \mathcal{T}$ **and** $\frac{t_{min}}{t_{max}} < 1 - \epsilon$ **then**
2. $d \leftarrow \epsilon.\bar{t}$
3. **while** $t_{max} - t_{min} > d$ **do**
4. **Let** $S_{max}, S_{min} \in \mathbf{S}$ **such that**
5. $t_{S_{max}} = t_{max}$ **and** $t_{S_{min}} = t_{min}$
6. $F \leftarrow$ REMOVE-FRAGMENT$(S_{max}, \frac{t_{max} - t_{min}}{2})$
7. ADD-FRAGMENT(S_{min}, F)
8. Recomputing $t_{max}, t_{min}, t_{S_{max}}, t_{S_{min}}$
end

REMOVE-FRAGMENT(S, d)
begin
1. **if** $\exists F \in S :\ t_F < d$ **then**
2. $S = S \setminus \{F\}$
3. **return** F
4. **else**
5. **Let** F **be** a fragment of S
6. **return** SPLIT-FRAGMENT(F, f)
where $f = \lfloor \frac{nodenum(F).d}{t_F} \rfloor$
end

6. Conclusion

In this paper, we have introduced and studied several fundamental problems of parallel processing search engine for very large XML data sets. The XML tree is fragmented into fragments, which are stored in sites. The fragments can be split, unified or move from some site to the other. Our querying language is a class of regular queries, and support string pattern matching. Our query processing is based on partial evaluation, which allow each site process the query in parallel. The redundant operations are rejected by preprocessing over the tree-index. We manage the fragmentation by our simple statistical system and the fragment operations. The fragmentation algorithm guarantees there is no big different processing cost between sites so that there is no overload in our system.

We have analyzed and done the experiments for the different query processing approaches over XML data sets in distributed environment. The results, which will be published soon, show that our algorithm overcomes the other algorithms both in theory and in experiment. We have no experiments for our fragmentation algorithm yet. Our plan is to analyze and do experiments for our fragmentation algorithm, and to implement a parallel search engine based on our ideas in future.

References

[1] Talwadker, A. S., Survey of performance issues in parallel database systems, In *Journal of Computing Sciences in Colleges archive*, Vol. 18, Issue 6, (2003).

[2] Suciu, D., Distributed query evaluation on semistructured data, *ACM Transactions on Database Systems*, Vol. 27 , Issue 1, (2002).

[3] Buneman, P., Cong, G., Fan, W., Kementsietsidis, A., Using partial evaluation in distributed query evaluation, In *Proceedings of the 32nd international conference on Very large data bases*, Vol. 32, VLDB (2006).

[4] Berglund, A., Boag, S., Chamberlin, D., Fernandez, M. F., Kay, M., Robie, J., Simeon, J., XML path language (xpath) 2.0. (August 2002), `http://www.w3.org/TR/xpath20`

[5] Boag, S., Chamberlin, D., Fernandez, M. F., Florescu, D., Robie, J., Simeon, J., XQuery 1.0: An XML Query Language, (November 2006), `http://www.w3.org/TR/xquery/`

[6] Deutsch, A., Fernandez, M., Florescu, D., Levy, A., Suciu, D., A query language for XML. In Proceedings of the Eights International World Wide Web Conference (WWW8), Toronto.