Efficient Processing Regular Queries In Shared-Nothing Parallel Database Systems Using Tree- And Structural Indexes *

Vu Le Anh, Attila Kiss

Department of Information Systems, ELTE University, Hungary
leanhvu@inf.elte.hu, kiss@ullman.inf.elte.hu

Abstract. In this paper, we introduce and study an efficient regular queries processing algorithm on a very large XML data set which is fragmented and stored on different machines. The machines are connected by the high speed interconnection. In this system the efficiency of a query processing algorithm depends on two main factors: the *waiting time* for the answer and the *total query processing and communication cost* over all machines of the system. In the partial processing approach, the query is sent to and partially evaluated at each server in parallel. The parallelism reduces the waiting time, but there are several redundant operations as it has to compute all possible cases for each fragment. In the stream processing approach, the query processing cost is minimized by parsing the data graph with the query. A fragment is visited if it is necessary, but there is no parallelism and the communication cost is high. To take the advantages of the shared-nothing parallel system, our algorithm is based on the partial evaluation. We describe two types of redundant operations. They are rejected by pre-computing the query on our tree- and structural indexes. The sizes of the indexes and the processing costs over them are considered as constants. Our algorithm overcomes two above algorithms according both the waiting time and the total query processing and communication cost criteria both in theory and in experiment.

1 Introduction

There are two motivational factors in the causation of the environment described in this paper: *Shared-nothing Parallel Database System* [1] and *XML* [13]. Our concept of shared-nothing parallel database system is the system of machines connected by high speed interconnection. The parallel database systems are being used in a wide variety of systems as they are extensible and are powered by the growth of RAM volumes and the speed of local network 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,

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

Y. Ioannidis, B. Novikov and B. Rachev (Eds.): Local Proceedings of ADBIS 2007, pp. 70-85

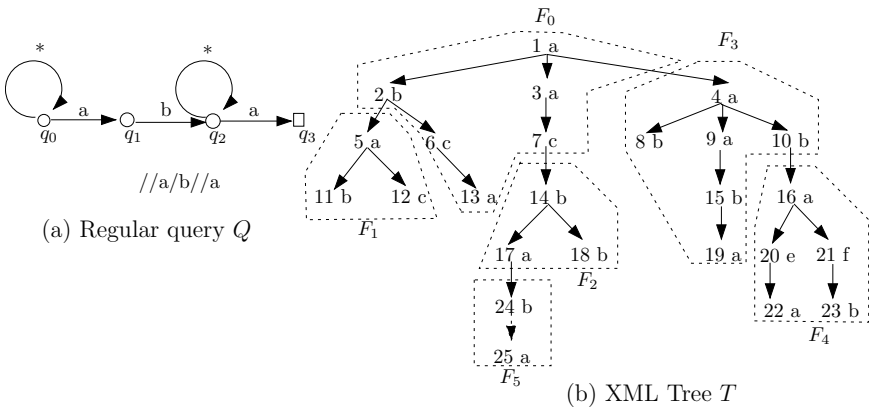

Fig. 1. An Example of fragmented XML tree and regular query

especially in self-describing, extendibility and inter-operability. XML is a good choice for representing various types, dynamic schema or open data sets. In this paper, we work with the shared-nothing parallel database system which is considered as a very large XML data set stored and fragmented over machines.

To retrieve XML and semi-structured data, several query languages have been proposed (XML-QL [10], UnSQL [9], XPath [11], and XQuery [12]). The common feature of these languages are the use of *regular path expressions*. A data node is an element of the answer of a *regular query*, if the path from the root to this node matching the corresponding regular path expression. This paper introduces and analyzes efficient techniques for processing regular queries in shared-nothing parallel database system using tree and structural indexes. The criteria of the efficiency are: the *waiting time* for the answer and the *total query processing and communication cost* over all machines of the system. There are two approaches for processing regular queries in distributed environment: *stream processing* and *partial processing*. We will study and compare our algorithm with the *partial parallel processing* algorithm represented for partial processing approach [5, 4] and the *tree traversal* algorithm represented for stream processing approach [5, 4].

As an example, we consider the regular query $E = //a/b//a$ (written in XPath syntax) over the XML tree T depicted in Fig.1(b). T is fragmented by $F_0, \ldots, F_5$. E can be represented by the nondeterministic finite automata Q depicted in Fig. 1(a). The answer of E over T is the set of data nodes $A = \{5, 13, 16, 19, 22, 25\}$. A is determined by traversing T and Q with following rules: (1) beginning at $(1, q_0)$ (the pair of the roots of T and Q) ; (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 A if some (u, q) is visited and there exists a transition from state q to final state p with the label of u.

In fragmented environment, the above algorithms works differently as follows:

Tree traversal. The traversal algorithm is modified as follows. In rule (2), if u is the root of another fragment then we jump to and continue processing over the sub-fragment. After finishing the process the sub-fragment will give back the control and the result to the parent fragment and the parent fragment will continue processing. In the example, with the depth-first traversal the the list of visited fragments sorted by time is $F_0F_1F_0F_1F_0F_2F_5F_2F_5F_2F_0F_3F_4F_3F_4F_3F_0$.

Partial Parallel Processing. The basic processing operation (F, q) is to process over a fragment F beginning at the root of F with state q without communicating other sites or operations. If the operation (F_2, q_3) is processed, there is no communication with the operations over F_0 or F_5 and the result is $\{17\}$. We process each site in parallel. At each site, we compute all possible processing operations and find out the relationship of the operations. The relationship of operations tells us about which operations are processed over the child-fragment if some operation is processed over the parent-fragment. In the example, in the site containing F_2 we process four operations $(F_2, q_0), \ldots, (F_2, q_3)$ over F_2 and we will find out following facts of relationship of the operations:

if (F_2, q_0) *is processed,* (F_5, q_0) *and* (F_5, q_1) *will be processed*
if (F_2, q_1) *is processed,* (F_5, q_2) *will be processed*
if (F_2, q_2) *is processed,* (F_5, q_2) *and* (F_5, q_2) *will be processed*

Based on all map of operations from the sites the chosen master site will decide which operations are reachable if the operation (F_{root}, q_{root}) is processed (F_{root} : the fragment containing the root of the tree; q_{root}: is the start state of the query graph). The reachable operations in the 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). Finally, each site only sends the sub-results of reachable operations, which has been computed in previous phases, to the master site. The result of the query is the union of them.

There is no parallelism in the tree traversal algorithm so the waiting time is not reduced. Moreover the communication cost is high as each fragment and each site may be visited by many times (F_0 is visited 5 times, F_1 is visited 2 times, etc). With the partial parallel processing algorithm, the waiting time is reduced by the parallelism. The communication cost is also reduced as each site is visited only one time, and the total network traffic is independent from the size of the XML tree [4]. However, the total processing cost may be high as for each fragment we process for all possible operations. This is not good in case the system serves many clients in the same times.

Our first contribution is to introduce the indexes. The tree-index stores information of the paths connecting the roots of fragments. The *DL-tree-indexes* coming from the *DL-indexes* [7] summarize the fragment structure. The total size of the indexes depends only on the number of fragments, which is considered as constant in shared-nothing parallel database system. Our second contribution is an efficient algorithm for processing regular queries over shared-nothing parallel database system. We define two types of redundant operations in the partial processing approach. The redundant operations of type 1 are the operations which are not reachable from the operation (F_{root}, q_{root}). The redundant operations

type 2 are the operations which do not find out any matching nodes. The redundant operations type 1 are determined by processing the query over the tree index . The redundant operations type 2 are restrained by processing over some structural index of the fragment. Our algorithm firstly processes on the indexes to reject redundant operations. The processing cost is considered as constant in case the total size of the index is considered constant. The pre-computing on the indexes not only decreases the total query processing in the partial evaluation phrase but also minimizes the communication cost and the waiting time as sites communicate only *one time* with the master site and processing with minimal number of redundant operations. Our third contribution is an experimental study evaluating our algorithm versus other approaches. Our experimental results show that our algorithm overcomes the other algorithms according both the waiting time and the total query processing and communication cost.

The remainder of the paper is organized as follows. Section 2 is the preliminary. We introduce the concepts of the data model and the regular queries and a template for processing over a fragment using in this paper. Section 3 is about stream processing vs. partial processing approaches. We study two algorithm: tree traversal algorithm represented for stream processing approach and partial parallel processing algorithm represented for partial processing approach. In section 4, we introduce and describe the technics to using tree- and structural indexes to processing regular queries over share-nothing parallel database system. In section 5, we present our experiments. Section 6 is the related works. Section 7 concludes the paper.

2 Preliminaries

2.1 Data Model

The data set is modelled as a *labelled tree* T. Σ is an alphabet. Each node v of T is labelled by a label value $label(v) \in \Sigma$. T is fragmented by a collection $\mathbf{F}$ of disjoint trees, or *fragments* F_i. The fragments are stored in different *sites* (or machines). 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 Fig. 1, this is fragment F_0. Given two fragments F and F', we say that F is a *child-fragment* of F' and 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 Fig. 1, fragment F_5 is a child-fragment of F_2 and F_0 is a parent-fragment of F_1 . $site(F)$ denotes the site containing fragment F. Naturally, we assume that $site(F) \neq site(F')$ if F is a child-fragment of F'.

2.2 Regular Queries

A regular expression over Σ alphabet is defined as follows:

$$R = \ \varepsilon |\ a \mid R_1.R_2 \mid R_1|R_2 \mid R_1^*$$

where ε is the empty word; $a \in \Sigma$ is a letter; R_1, R_2, R are regular expressions; $(.)$, $(|)$, $(*)$ are the concatenation-, the alternation-, and the iteration operations respectively. Each regular query is defined by a regular expression, and each regular expression also defines a regular language over Σ, which contains all words over Σ marching the regular expression. We call the rooted labelled directed graph of the finite nondeterministic automata, which computes the regular language defined by the regular expression, the *query graph* of the query. The start state of the automata is denoted by q_{root}. A data node u is an element of the result of regular query Q over tree T, if the label of the path from the root to u matches the according path expression. You can see the example of query graph in Fig. 1. We just remark our readers that the formula writing the query in the example is in XPath syntax [11], which is converted easily to our defined syntax.

2.3 Processing Over Fragment

We introduce and study basic operation processing a fragment begin at the root with some state of the query graph. The template of the operation processing fragment F with state q is as follows:

```
FRAGMENT-PROCESS(F,q)
begin
1. FragResult ← ∅
2. PRE-PROCESS(F,q)
3. SCAN(root(F),q)
4. POST-PROCESS(F,q)
5. return FragResult
end

MATCHING(q,label)
begin
1. S ← ∅
2. for each transition (q, label, p) do
3.    S ← S ∪ {p}
4. return S
end

SCAN(u,q)
begin
1. if u is stored in another fragment then
2.    WORKING-LINK-NODE(u,q)
3. else
4.    for each state p ∈ MATCHING(q, label(u)) do
5.       if p is final state then
6.          FragResult= FragResult ∪ {u}
7.       for each edge (u, v) do
8.          SCAN(v,p)
end
```

In the generalized fragment processing algorithm, the MATCHING(q,*label*) procedure returns the set of states p where there exists a transition from state q to state p labelled by *label*. We traverse the data graph with the SCAN procedure. If a pair (u, q) is visited by calling SCAN(u,q), these pairs (v, p) are also visited for each edge (u, v) and $p \in$ MATCHING(q,*label*(u)). Each pair (u, q) is visited maximum one time as T is a tree. Therefore, the SCAN procedure is never in infinity loop. The WORKING-LINK-NODE procedure is called when we meet a node stored in another fragment. The main differences between the query processing algorithms over distributed environment are presented by the implementation of WORKING-LINK-NODE, PRE-PROCESS and POST-PROCESS virtual procedures for working with nodes stored in another fragment, doing before or after processing fragment respectively.

3 Stream Processing vs. Partial Processing Approaches

We study two most popular approaches for processing regular queries in distributed environment: stream processing and partial processing. We study tree traversal algorithm represented for stream processing algorithm approach in subsection 3.1 and partial parallel processing algorithm represented for partial processing approach in subsection 3.2.

3.1 Tree Traversal Algorithm

The main behavior of stream processing approach when we meet a node stored in another fragment is to jump to the new fragment for continuing processing. The parent process waits until the process operation over the child-fragment finishes, then it receives the result of the operation from the child-fragment and continue processing. Therefore, the POST-PROCESS and WORKING-LINK-NODE procedures are implemented as follows:

POST-PROCESS(F,q)
begin
1. **if** $F \neq F_{root}$ **then**
2. SEND-RESULT($Result$)
end

WORKING-LINK-NODE(u,q)
begin
1. Let F be the remote fragment containing u
2. FRAGMENT-PROCESS(F,q) (Calling from remote)
3. $subResult \leftarrow$RECEIVE-RESULT(F)
4. $FragResult \leftarrow subResult \cup FragResult$
end

The SEND-RESULT procedure is used for sending the result to the parent-fragment. The RECEIVE-RESULT(F) procedure is used for receiving the result from the operation over child fragment F. A RECEIVE-RESULT procedure calling synchronizes with some SEND-RESULT procedure calling. Finally, the result of the query on tree T is the result of the fragment processing operation (F_{root}, q_{root}). It is determined by calling the SPIDER procedure.

SPIDER(Q)
begin
1. **return** FRAGMENT-PROCESS(F_{root},q_{root})
end

The example of the tree traversal algorithm is shown in section 1. The tree traversal algorithm can be considered as a sequence of calling the FRAGMENT-PROCESS procedure. There is no parallelism. Moreover each site and fragment can be visited many times and the number of the communication between sites is not constant, it depends on the query graph. Therefore, the waiting time for the answer is very high, and it does not take the advantages of the share-nothing parallel database systems.

3.2 Partial Parallel Processing Algorithm

The main behavior of the partial processing approach is to process each operation independently. There is no communication with other operation while processing. When we meet some node stored in another fragment, we just write down the relationship between the current operation and the corresponding operation over the child fragment. Therefore, the PRE-PROCESS and WORKING-LINK-NODE procedures are implemented as follows:

```
PRE-PROCESS(F,q)                WORKING-LINK-NODE(u,q)
begin                           begin
1. bF ← F                       1. Let F be the remote fragment containing u
2. bQ ← q                       2. Map ← Map ∪ ((bF, bQ), (F, q))
end                             end
```

The fact *"if operation* (bF, bQ) *is processed then* (F, q) *will be processed"* is coded by $((bF, bQ), (F, q))$. (bF, bQ) is current operation. bF and bQ variables are initialized in the PRE-PROCESS procedure. Map is the global variable storing all relationships of operations we find out when processing over the site. The query processes over sites are independent and in parallel. At each site, all possible operations are processed firstly. After that the map of operations is sent to the chosen *master site*. Then each site waits until the master site sends the set of operations, which are reachable when we process the operation (F_{root}, q_{root}). Finally, all sites just send the reachable results to the master site and the union of these result is the result of the answer. MASTER(Q) procedure is run in the master site to control the sites processing query graph Q. The partial result of query graph Q is computed in site S by using the SITE-PROCESS(S,Q) procedure. The reachable operations are determined by the COMPUTE-REACHABLE-RESULTS procedure on the master site. In the MASTER procedure, $Map[S]$ is the map of the operations we receive from site S and $RO[S]$ is the set of reachable operations of site S.

```
MASTER(Q)
begin
1. for each site S do in parallel
2.    SITE-PROCESS(S,Q) (Calling from remote)
3.    Map[S]← RECEIVE-MAP(S)
4. joint: waiting until receiving all map result
          from every site
5. COMPUTE-REACHABLE-OPERATIONS()
6. for each site S do in parallel
7.    SEND-REACHABLE-OPERATIONS(S,RO[S])
8.    subResult[S] ←RECEIVE-RESULT(S)
9. joint: waiting until receiving all result
          from every site
10. Result ← ∅
11. for each site S do
12.   Result ← Result ∪ subResult[S]
end
```

```
COMPUTE-REACHABLE-OPERATIONS()
begin
1. for each site S do
2.    RO[S] ← ∅
3. Stack ← ∅
4. Stack.push((F_root, q_root))
5. while Stack ≠ ∅ do
6.    (F, q) ← Stack.pop()
7.    Let S be the site containing F
8.    RO[S] ← RO[S] ∪ (F, q)
9.    for each ((F, q), (F', q')) ∈ Map[S] do
10.      Stack.push((F', q'))
end
```

```
SITE-PROCESS(S,Q)
begin
1. Map ← ∅
2. for each fragment F of S do
3.    for each state q of Q do
4.       result[F, q]←FRAGMENT-PROCESS(F,q)
5. SEND-MAP(Map)
6. RO←RECEIVE-REACHABLE-OPERATIONS()
7. Result ← ∅
8. for each (F, q) ∈ RO do
9.    Result ← Result ∪ result[F, q]
10.SEND-RESULT(Result)
end
```

The sequence of the activities ordered by time when processing a query graph Q is:

1. *Activating the query process over each site* (line 1-2 of the MASTER procedure).
2. *Computing the partial result at each site* (line 1-4 of the SITE-PROCESS procedure).
3. *Synchronization: Sending the map of the result from some site to the master site* (line 5 in the SITE-PROCESS procedure, line 3 in the MASTER procedure).
4. *The master waits until receiving all map results* (line 4 in the MASTER procedure).
5. *Determining reachable operations*(line 5 in the MASTER procedure).
6. *Synchronization: Sending the reachable results to each site (line 6 in the* SITE-PROCESS *procedure, line 7 in the* MASTER *procedure).*
7. *Each site determines the site result* (line 7-9 in the SITE-PROCESS procedure).
8. *Synchronization: Sending the result from some site to the master site* (line 10 in the SITE-PROCESS procedure, line 8 in the MASTER procedure).
9. *The master waits until receiving all site results* (line 9 in the MASTER procedure).
10. *Determining the final result* (line 10-12 in the MASTER procedure).

The waiting time for the answer is reduced by the parallelism. Each site and fragment is visited only one time. Moreover the number of the communication between sites is constant, and the size of communicated data is only depend on the size of the query and the result [4]. We can reduce the waiting time by expanding the system with more machines (sites). The total processing cost is higher than the necessity as we process for each fragment with all possible states.

4 Processing Queries with Tree- and Structural Indexes

We begin this section by defining two types of redundant fragment processing operations in partial processing approach.

Definition 1. *(F, q) is redundant type 1 if it is not reachable when we process the operation (F_{root}, q_{root}). (F, q) is redundant type 2 if there is no data node in the fragment F matching the query when we process the operation.*

In the example in Fig. 1, (F_2, q_2) is a redundant operation type 1 but not a redundant operation type 2 as if from all operations over F_2, (F_2, q_0) is the only

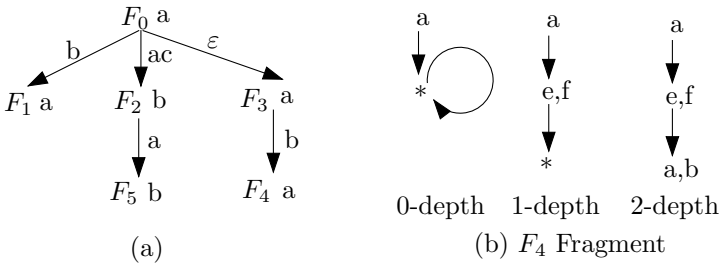

Fig. 2. Tree and structural indexes of the tree T in Fig. 1(b)

operation which is reachable from (F_0, q_0). Furthermore, if (F_2, q_2) is processed, node 18 matches the query. In another case (F_4,q_0) is a redundant operation type 2 but not a redundant operation type 1. Obviously, we do not need to carry out an operation if it is redundant type 1 or 2.

4.1 Tree Index

Definition 2. *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 labelled by the root of fragment F*
(ii) (F, F') is an index edge if F is parent fragment of F'. (F, F') is labelled 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.

The tree index stores the information about the paths connecting the roots of the fragments. The index tree of the fragmented tree T in Fig 1 is depicted in Fig 2(a). In this tree, (F_0, F_2) is labelled by $a.c$ which is the label path of the path $< 3, 7 >$; (F_0, F_3) is labelled by ε as the root of F_0 is the parent of the root of F_3. The size of the index tree can be considered constant as the number of fragments is considered constant.

The (F, q) operations which are not redundant type 1 are determined by the PROCESS-INDEX-I procedure as follows:

PROCESS-INDEX-I(T_I,Q)
begin
1. $RO \leftarrow \emptyset$
2. $Stack \leftarrow \emptyset$
3. $Stack.push((F_{root}, Root(Q)))$
4. **while** $Stack \neq \emptyset$ **do**
5. $(F, q) \leftarrow Stack.pop()$
6. $RO \leftarrow RO \cup (F, q)$
7. **for each** index edge (F, F') **do**
8. **for each** $q' \in$MATCHING-II$((F, F'), q)$ **do**
9. $Stack.push((F', q'))$
10.**return** RO
end

MATCHING-II((F, F'),q)
begin
1. $S \leftarrow$ MATCHING$(q, label(Root(F)))$
2. Let $< l_1, \ldots, l_k >$ be the label of (F, F')
3. **for** $i = 1$ **to** k **do**
4. $S' \leftarrow \emptyset$
5. **for each** state $q \in S$ **do**
6. $S' \leftarrow S' \cup$ MATCHING(q,l_i)
7. $S \leftarrow S'$
8. **return** S
end

The structure of PROCESS-INDEX-I procedure is similar to the COMPUTE-REACHABLE-RESULTS procedure's in the partial parallel processing algorithm. The difference between two procedures is the way expanding new reachable operations. We use the MATCHING-II procedure (at line 8) in the first procedure and $Map[S]$ (at line 9) and in the second. Both of them give the same information that *"Which operations over the child-fragment are reachable if some operation over parent-fragment is reachable."*. The correctness of the MATCHING-II can be proven easily as it is considered as a sequence of applying the MATCHING procedure. Therefore, the PROCESS-INDEX-I procedure will return the set of operations which are reachable when we process (F_{root}, q_{root}). These operations are not redundant type 1. The processing cost over tree index is considered as constant as the size of the tree index is considered as constant.

4.2 Structural Indexes

We restrain redundant operations type 2 by processing the query graph over structural indexes of the fragments. In general, a structural index $I_{\mathcal{G}}$ of the graph $\mathcal{G}$ is built by the following general procedure: (1) partitioning the data nodes into classes according to some equivalence relation, (2) making an index node for each equivalence class and the index node is labelled by the union of the labels of the data node elements, and (3) adding an index edge from index node I to index node J if there exists an edge from data node u to data node v, where u is an element of I and v is an element of J. In the case $\mathcal{G}$ is a rooted graph, the index node $root(I_{\mathcal{G}})$ containing the root of $\mathcal{G}$ is defined as the root of $I_{\mathcal{G}}$.

The traversal rules for processing operation ($I_{\mathcal{G}}$,q) over the rooted index graph $I_{\mathcal{G}}$ beginning with state q as follows: (1)We begin at ($root(I_{\mathcal{G}})$,q) .(2) From (I, p) we visit (J, p') for each index edge (I, J) and there exists a transition from state p to state p' with the label $l \in label(I)$. (3) I matches the query if some (I, p) is visited and there exists a transition from state p to final state p' with label $l \in label(I)$.

Proposition 1. *Let $\mathcal{I}$ be a structural index of fragment F. (F, q) is a redundant operation type 2, if there is no index node matching the query when we process ($\mathcal{I}$,q) operation.*

Proof. We use indirect method. We assume (F, q) is not a redundant operation type 2. It implies that there exists a path $u_1u_2 \dots u_n$ and a sequence of states $q_0q_1 \dots q_n$ such that (i) u_1 is the root of F and $q_0 = q$ (ii) u_i are data nodes of F (iii) q_n is final state (iv) there is a transition from state q_{i-1} to state q_i with label $label(u_i)$ $(i = 1, \dots, n)$. Let $U_1, \dots, U_n$ be the index nodes of $\mathcal{I}$ containing $u_1, \dots, u_n$ respectively. The construction of the index implies that (v) U_1 is the root of $\mathcal{I}$ (vi) (U_i, U_{i+1}) are index edges $(i = 1, \dots, n-1)$. $label(u_{i-1}) \in label(U_{i-1})$ and there exists a transition from state q_{i-1} to state q_i with label $label(u_i)$ so we can traverse from (U_{i-1}, q_{i-1}) to (U_i, q_i). It implies that U_n matching the query we traverse the query graph beginning at the root of $\mathcal{I}$, U_1, and state $q_0 = q$ (conflict).

The proposition gives us a necessary condition to check if (F, q) is a redundant type 2. The cost for checking the condition depends on the size of the structural index, which is not bigger than the fragment. The condition is sufficient if the structural index is the 1-index [14]. However, the size of the 1-index is big in most cases. The size of the structural index should be constant. Based on the DL-indexes [7] we introduce and study the *DL-tree-indexes* using for checking redundant operations type 2.

The processing cost over a structural index depends on the number of the labels of the data nodes in each index nodes. We define special label $'*'$ ($* \notin \Sigma$) for matching any label of Σ. If the label of an index node I is $'*'$, we can jump from (I, q) to any (J, p) where (I, J) is an index edge and there exists a transition from state q to state p.

Definition 3. *Let F be the fragment, h be the depth of F and k be an integer $1 \leq k \leq h$. The k-depth DL-tree-index of F, $\mathcal{I}_k$, is a structural index in which the partition of nodes is as follows:*

(i) For each $0 \leq l \leq k$ we group the data nodes in the same depth l in the same index node I_l.

(ii) In the case $k < h$ we group the data nodes whose depth is more than k in the same index node and the index node I_{k+1} labelled by $'*'$.

The DL-tree-indexes of fragment F_4 of tree T in Fig. 1(b) is shown in Fig. 2(b). The root of $\mathcal{I}_k$ is I_0. The $\mathcal{I}_k$ has k index nodes in the case $k = h$ otherwise $k + 1$ index nodes. We can choose k value as bigger as the size of $\mathcal{I}_k$ can be considered as constant. The process over a structural index $\mathcal{I}_k$ of the fragment F to determine if (F, q) is a redundant type 2 is as follows:

```
PROCESS-INDEX-II(I_k,q)
begin
1. S ← {q}
2. for i = 0 to k do
3.    S' ← ∅
4.    for each state p ∈ S do
5.       for each label l ∈ label(I_i) do
6.          S' ← S' ∪ MATCHING(p,l)
7.    if EXISTS-FINAL-STATE(S') then
8.       return true
9.    S ← S'
10.if ∃ I_{k+1} ∧ S ≠ ∅ then
11.   if MATCHING-III (I_{k+1}, S) then
12.      return true
13. return false
end
```

```
EXISTS-FINAL-STATE(S)
begin
1. for each state p ∈ S do
2.    if p is final state then
3.       return true
4. return false
end

MATCHING-III(I, S)
begin
1. if ∃ index edge (I, I) then
2.    return true
3. for each state p ∈ S do
4.   if ∃ transition (p, p') and p' is final state then
5.    return true
6. return false
end
```

Proposition 2. *If the* PROCESS-INDEX-II*($\mathcal{I}_k$,q) returns* **false** *value, (F, q) is a redundant operation type 2.*

Proof. Let $S_0, \ldots, S_k$ be the values of S at the beginning of the loop at line 2 for each $i = 0, \ldots, k$ respectively. We will show that *"S_i $(i = 0, \ldots, k)$ is the set of states p_i, in which (I_i, p_i) is visited if $(\mathcal{I}_k, q)$ is processed"* (1). Obviously (1) is true for $i = 0$ as $S_0 = \{0\}$. The S_{i+1} computation based on S_i at line 4-6 guarantees for the correctness of the proof for (1) by induction.

The return value is **false** so the condition we check at line 7 and line 11 must be false. EXISTS-FINAL-STATE(S_i) returning **true** value (at line 7) implies that S_i $(i = 1, \ldots, k)$ not containing any final state. Therefore, (1) implies that I_i $(i = 0, \ldots, k)$ not matching the query. MATCHING-III (I_{k+1}, S) returning **false** value (at line 11) implies that I_{k+1} does not match the query either. No index node matches the query when we process $(I_\mathcal{G}, q)$, so (F, q) is a redundant operation type 2.

The DL-indexes [7] are quite dynamic and flexible. We can use a DL-tree-index as the initial index graph and then refining the index so that the size is fit with the system or the index supports a set of frequently queries [7]. The processing index over the refining index should be modified so that avoiding the circles in the index graph. However, the processing cost is constant if the size of the index is considered as constant.

4.3 Processing Queries with Tree- and Structural Indexes

We store the tree index T_I of the system and for each fragment F we also store $Index(F)$ which is a k-depth DL-tree-index of F on the chosen master site. The EFF-MASTER(Q) procedure is run in the master site to control the sites processing query graph Q. The partial result of query graph Q is computed in site S by using the EFF-SITE-PROCESS(S,Q) procedure.

EFF-MASTER(Q)
begin
1. *States* ← PROCESS-TREE-I(T_I,Q)
2. **for each** site S **do**
3. $RO[S] \leftarrow \emptyset$
4. **for each** $(F, q) \in States$ **do**
5. **if** PROCESS-INDEX-II($Index(F)$,q) **then**
6. Let S be the site containing F
7. $RO[S] \leftarrow RO[S] \cup (F, q)$
8. **for each** site S **do in parallel**
9. EFF-SITE-PROCESS(S,Q) (Calling from remote)
10. SEND-REQUIRED-OPERATIONS(S,$RO[S]$)
11. $subResult[S]$ ←RECEIVE-SITE-RESULT(S)
12.**joint:** waiting until receiving all result from every site
13.$Result \leftarrow \emptyset$
14.**for each** site S **do**
15. $Result \leftarrow Result \cup subResult[S]$
end

EFF-SITE-PROCESS(S,Q)
begin
1. RO←RECEIVE-REQUIRED-OPERATIONS()
2. $Result \leftarrow \emptyset$
3. **for each** $(F, q) \in RO$ **do**
4. $Result \leftarrow Result \cup$FRAGMENT-PROCESS$(F, q)$
5. SEND-SITE-RESULT($SiteResult$)
end

The sequence of the activities of the query process ordered by time is:

1. *Rejecting the redundant operations at master site*: (line 1-7 of the MASTER procedure).
2. *Activating the query process at each site* (line 9 in the EFF-MASTER procedure)
3. *Synchronization: Sending the required operations to each site* (line 10 in the EFF-MASTER procedure, line 1 in the EFF-SITE-PROCESS procedure).
4. *Computing the required operations at each site.* (Line 3-4 in the EFF-SITE-PROCESS procedure)
5. *Synchronization: Sending the site result from some site to the master site* (line 5 in the EFF-SITE-PROCESS procedure, line 11 in the EFF-MASTER procedure).
6. *The master waits until receiving all site results* (line 12 in the EFF-MASTER procedure).
7. *Determining the result* (line 13-15 in the EFF-MASTER procedure).

Obviously, the number and the cost of communication in our algorithm is smaller than two algorithm in section 3. The total cost of query processing in each site is also smaller as the number of processing operations is fewer. If the number of fragments is considered constant we can ignore the cost of processing over tree- and structural indexes. In this case our algorithm overcomes two above algorithms according both the waiting time and the total query processing and communication cost criteria.

5 Experiments

We compared the performance of our efficient algorithm using tree and structural indexes (**EPP** for short), the partial processing algorithm (**PP** for short) and the tree traversal algorithm (**TP** for short) in the waiting time and the total processing and communication criteria. We implemented the algorithms in C++ programming language.

Data set and fragmentation. We used the *XMark* data set containing the activities of an auction Web site is generated by using the Benchmark Data Generator [18]. The size of our data set is about 500 Mb. The number of nodes is 10 267 360 and the number of labels is 77. We uses 19 Linux machines over a local LAN. We use a program splitting the XML tree into 76 fragments. Each fragment has about 150 000 nodes. The fragments are chosen randomly to be stored in the sites.

Queries. We proposed 10 regular path queries representing for different conditions of the environment for our experiments. They are:

1. **Q1:** *//keyword.*
2. **Q2:** *//listitem/text/keyword/bold.*
3. **Q3:** *//asia/item//keyword/bold.*
4. **Q4**: *//listitem//keyword*
5. **Q5**: *//people/*/profile/income*
6. **Q6**: *//bold | //emph*
7. **Q7**: *//asia//bold | //asia//emph*
8. **Q8**: *//person//name*
9. **Q9**:*//item/mailbox/mail*
10. **Q10**:*//namerica//description//keyword*

We processed these queries with our algorithms many times and measured the average values of the waiting time and the cost of processing and communication for each query and each algorithm. Here are the results:

	Waiting time(ms)			Pro. & Com. time(ms)		
Queries	TP	PP	EPP	TP	PP	EPP
Q1	138662	4590	3845	33990	10843	10823
Q2	128900	1987	1765	8760	8780	4529
Q3	83516	2638	1781	7993	16111	7812
Q4	13731	3969	2682	13490	18153	10326
Q5	85461	2093	1910	9780	8677	8360
Q6	136715	6921	6192	59970	14459	14321
Q7	87661	12673	2045	12081	25519	8660
Q8	88925	3208	1843	52560	18380	13209
Q9	117297	7942	1641	40773	14893	5132
Q10	91085	4174	2204	15420	28183	9518

Waiting time. The ratio of the waiting time of three algorithms, **EPP : PP : TP**, is **1 : 1.94 : 37.52**. Our experiments shows that the partial processing approach absolutely overcomes the stream processing approach according to the waiting time criterion. There are two reasons for this explanation: (1) **TP** does not use the parallelism like **PP** and **EPP**; (2) there are many communication between sites. In cases **Q3, Q7** and **Q10**, although the processing and communication cost of **TP** is lower than **PP**'s but the waiting time is still higher. **EPP** overcomes **PP** according the waiting time criterion as the ratio is about 50% and being better in all test cases. Especially in cases **Q7** and **Q10**, there are a lot of redundant operations for **PP** but they are restrained by preprocessing over indexes in **EPP**.

Processing and Communication Cost. The ratio of the processing and communication cost of three algorithms, **EPP : PP : TP**, is **1 : 1.77 : 2.75**. The communication cost makes the cost of **TP** is higher than **PP** in general. However the redundant operations types 1 make the total cost of **PP** is higher than **TP** in several cases (**Q3, Q7** and **Q10**). By restraining redundant operations, the cost of **EPP** is always the lowest in all test cases among three algorithms.

Our experiments shows that our **EPP** algorithm is the best among the introduced algorithms according the waiting time and the processing and communication cost criteria.

6 Related Works

The fundamental concepts of parallel database systems have been found in [1]. Architecture, data placement and query optimization are the focuses of interest. We use the shared-nothing architecture for our system with no data replacement. Some ideas and suggestions about the fragmentation (data placement) are introduced in [8].

The problem of optimizing, processing regular path expressions in the context of navigating semi-structured data in web sites has been introduced and studied in [2, 3]. The focus of [2] is on the use of local information expressed in the form of path constraints in the optimization of path expression queries. In [3] two query optimization techniques is proposed to rewrite a given regular path expression into another query that reduces the scope of navigation. [4, 5] are closely related to our work as they studied the distributed query evaluation on semi-structured data using partial evaluation. The boolean XPath queries [5] and the structural recursion queries [4] are introduced and studied. The main difference between these works and our work is the context of the system. All of them [2–5] assume that the data set is distributed in different machines in "weak" relationship and the communication cost may be very expensive. In their context the fragmentation is not manageable and the indexes describing the relationship between sites are not suggested. The spirit of the processing algorithms described in [4, 5] is similar to our partial parallel processing algorithm.

The structural indexes [6, 7, 14–17] are introduced to speed up the query evaluation over semi-structured data by simulating the data graph by an index graph. An index graph is equivalent with the data graph over a given query if the processing the query over the index graph and the query graph bring the same result. The 1-index [14, 6] is equivalent with the data graph over all regular queries. The A(k)-index [15] is equivalent with the data graph over all primary queries not longer than k. The D(k)-index [16], the M(k)-index [17] and DL-indexes [7] are equivalent with the data graph over a given set of primary queries. In our context, the size of the structural indexes simulating the fragments are considered constant so the chosen indexes must be dynamics and adaptive. Our DL-tree-index are improved from the DL-indexes, which are the most adaptive and dynamics indexes.

7 Conclusion

We have shown an efficient algorithm processing regular queries in shared-nothing parallel database systems based on partial evaluation. We define two types of redundant operations. The redundant operation type 1, which are not reachable, are determined by the tree structural. The redundant operations type 2, which have no matching node, are restrained by processing over structural indexes. The DL-tree-indexes are introduced and proposed for this purpose. The size of the tree index and the DL-tree indexes are considered as constants. The partial evaluation becomes more effective by restraining redundant operations. Our experimental study has verified the effectiveness of our techniques.

We plan to extend the current work in a number of directions. First, the effectiveness of our algorithm depends on the fragmentation. An effective fragmentation algorithm concerns with the statics system and the set of frequency queries. Some ideas are introduced in our work [8]. Second, we plan to extend our algorithms to handle more general queries in XPath and XQuery. The partial evaluation, indexes for these queries are open problems.

References

1. Ameet S. Talwadker 2003. Survey of performance issues in parallel database systems. In Journal of Computing Sciences in Colleges archive Volume 18, Issue 6.
2. S. Abiteboul, V. Vianu 1997. Regular path queries with constraints. In Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems.
3. M. Fernandez and D. Suciu 1998. Optimizing regular path expressions using graph schemas. In Proceedings of the 14th Inter. Conference on Data Engineering.
4. D. Suciu 2002. Distributed query evaluation on semistructured data. ACM Transactions on Database Systems Volume 27 , Issue 1.
5. P. Buneman,G. Cong, W. Fan, A. Kementsietsidis 2006. Using partial evaluation in distributed query evaluation. In Proceedings of the 32nd international conference on Very large data bases Volume 32, VLDB 2006.
6. A. Kiss, V. L. Anh 2005. Combining Tree Structure Indexes With Structural Indexes. In Proceedings of 9th East-European Conference on Advances in Databases and Information Systems, LNCS 3631.
7. A. Kiss, V. L. Anh 2006. Efficient Processing SAPE Queries Using The Dynamic Labelling Structural Indexes. In Proceedings of 10th East-European Conference on Advances in Databases and Information Systems, LNCS 4512.
8. V. L. Anh, A. Kiss, Z. Vinceller 2007. Parallel Processing Search Engine For Very Large XML Data Sets. In ICAI 2007.
9. P. Buneman,M. Fernandez, and D. Suciu 2000. UNQL: A query language and algebra for semi-structured data based on structural recursion. In VLDB J.9, Issue 1.
10. A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu 1999. A query language for XML. In Proceedings of the Eights International World Wide Web Conference (WWW8), Toronto.
11. A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simeon. XML path language (xpath) 2.0. In http://www.w3.org/TR/xpath20, August 2002.
12. S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language. In http://www.w3.org/TR/xquery/, November 2006.
13. Extensible Markup Language (XML). In http://www.w3.org/XML/.
14. T. Milo and D.Suciu 1999. Index Structures for Path Expressions. In ICDT, 1999.
15. R. Kaushik, P. Shenoy, P. Bohannon and Ehud Gudes 2002. Exploiting Local Similarity for Efficient Indexing of Paths in Graph Structured Data. In ICDE,2002.
16. Q. Chen, A. Lim, K. W. Ong 2003. D(K)-Index: An Adaptive structural Summary for Graph-Structured Data. In ACM SIGMOD 2003, June 9-12.
17. Hao He, Jun Yang 2004. Multiresolution Indexing of XML for Frequent Queries. In Proceedings of the 20th International Conference on Data Engineering.
18. XMark: The XML benchmark project. http://monetdb.cwi.nl/xml/index.html.