WIN: An Efficient Data Placement Strategy for Parallel XML Databases

Nan Tang† Guoren Wang‡ Jeffrey Xu Yu† Kam-Fai Wong† Ge Yu†

† Department of SE & EM, The Chinese University of Hong Kong, Hong Kong, China
‡ College of Information Science and Engineering, Northeastern University, Shenyang, China
{ntang, yu, kfwong}@se.cuhk.edu.hk {wanggr, yuge}@mail.neu.edu.cn

Abstract

The basic idea behind parallel database systems is to perform operations in parallel to reduce the response time and improve the system throughput. Data placement is a key factor on the overall performance of parallel systems. XML is semistructured data, traditional data placement strategies cannot serve it well. In this paper, we present the concept of intermediary node INode, and propose a novel workload-aware data placement WIN to effectively decluster XML data, to obtain high intra query parallelism. The extensive experiments show that the speedup and scaleup performance of WIN outperforms the previous strategies.

1 Introduction

XML has become the de facto standard for data representation and exchange over the Internet. Nevertheless, along with the increasing size of XML document and complexity of evaluating XML queries, existing query processing performance in concentrative environment will get a limitation. Parallelism is a viable solution to this problem.

In parallel DBMSs, data placement has a significant impact on the overall system performance. Several earlier studies have shown that data placement is closely relevant to the performance of a shared-nothing parallel architecture. Many classical data placement strategies are thereby proposed in the literature. [2,11] proposed many single dimension and multiple dimension data placement strategies for structured data in RDBMSs. In OODBMSs otherwise, since the objects can be arbitrarily referenced by one another, the relationship of objects is regarded as a graph on topology. Thus most of the object-placement strategies are graph-based.

Distinct from them, XML employs a tree structure. Conventional data placement strategies in RDBMSs and OODBMSs cannot serve XML data well. Semistructured data involve nested structures; there are much intricacy and redundancy when representing XML as relations. Data placement strategies in RDBMSs cannot, therefore, well solve the data placement problem of XML data. Although graph partition algorithms in OODBMSs can be applied to tree partition problems, the key relationships between tree nodes such as ancestor-descendant or sibling cannot be clearly denoted on graph. Hence data placement strategies for OODBMSs cannot well adapt to XML data either. Furthermore, the path based XML query languages such as XPath and XQuery enhance the hardness and intricacy of tree data placement problem. Data placement strategies in both parallel relational and object-oriented database systems are helpful to the data placement for XML documents. They cannot, nevertheless, solve the problem which XML faces. Efficient data placement strategies for XML DBMSs are, therefore, meaningful and challenging.

[13] proposed two data placement strategies for XML data tree: NSNRR and PSPIB. NSNRR is proposed to exploit pipeline query parallelism. With NSNRR elements having different tag names are allocated to different sites in a round-robin way. Previous experiments show that when the document is small NSNRR has good speedup and scaleup performance; but as the document grows, both the speedup and scaleup performance of NSNRR behave badly. PSPIB is a data placement strategy based on path instances. Path instance is informally defined as a sequence of node identifiers from root to leaf node, while path schema is an abstract of path instances. The main idea of PSPIB is to decluster the path instances having
the same path schema evenly across all sites. Because PSPIB and WIN have the similar idea to partition the XML data tree to form sub-trees, we compare WIN with PSPIB in the performance evaluation. We give a simple example of data distribution of PSPIB to make it more concrete. Given a lightweight XML data tree in Fig. 1, Fig. 2 shows the data placement under the case of two sites. Solid circles represent the elements duplicated at both sites; real circles represent the elements distributed to site 1; dotted circles represent the elements distributed to site 2.

![Figure 1. A Lightweight XML Data Tree](image1)

![Figure 2. Data Placement of PSPIB](image2)

In this paper, we present the concept of intermediary node INode. Considering that determining the best data placement strategy in parallel DBMSs is an NP-complete [9] problem, a novel workload-aware data placement strategy WIN is proposed to find the suboptimal INode set for partitioning XML data tree. This strategy is not aim at special queries, nevertheless, as will be shown, this strategy suits each query expression for good intra query parallelism. Based on an XML database benchmark—XMark [10], comprehensive experimental results show that our new data placement strategy WIN has better speedup and scaleup performance than previous work.

The rest of this paper is organized as follows. Section 2 gives the concept of INode and the basic idea for data placement. Section 3 provides a detail statement of the novel data placement strategy WIN. Section 4 shows experimental results. Our conclusions and future work are contained in Section 5.

2 Data Placement using INode

Observe from a common XML data tree, we find that the number of upper level nodes is always few. And yet these nodes are the crux of many XML query expressions. Therefore, we should duplicate these nodes to all sites; the tiny storage space consumption can trade off between high intra query parallelism and low communication overhead. We will partition the remaining sub-trees to different sites, and then in each site, combine the sub-trees and duplicated nodes to form a local data tree. WIN describes how to find a group of duplicate nodes and a group of sub-trees to tradeoff the duplication cost and high parallelism. The problem is formally defined as follows:

Let $XT$ be an XML data tree, $Q$ be the query set of $n$ XML query expressions $Q_1, Q_2, \ldots, Q_n$, and $f_i$ be the frequency of $Q_i$, let $N > 1$ be the number of sites. How to partition $XT$ to $N$ sub-trees $XT_1 \sim XT_N$, $XT = \bigcup_{i=1}^{N} XT_i$, to take each $Q_i$ a high intra query parallelism.

We will first present the concept of INodes, and then give the basic data placement idea based on INodes, last we give our workload estimation formulae.

2.1 Intermediary Node

We classify nodes in an XML data tree $XT$ into three classes: Duplicate Node ($DN$), Intermediary Node ($IN$) and Trivial Node ($TN$). $DN$ will be duplicated at all sites. $IN$ is the essential node that decides which nodes will be partitioned to different sites. $TN$ will be partitioned to the site where their ancestor INodes is. We use DN to represent the set of $DN$es, IN for $IN$odes and TN for $TN$odes. They satisfy the following properties:

1. $DN$’s ancestor nodes are $DN$ or NULL;
2. $IN$’s ancestor nodes are $DN$ or NULL;
3. $IN$’s descendant nodes are $TN$ or attribute nodes or text nodes or NULL;
4. $DN$ is duplicated in all sites, $IN$, $TN$ are unique;
5. $IN \cap DN \cap TN = \emptyset$;
6. $IN \cup DN \cup TN = XT$.

$DN$ forms the duplicated root tree in all sites, and each INode in $IN$ is the root of a sub-tree to be partitioned. $IN$ is the pivot of our data placement strategy. For a certain $IN$, our partition algorithm will get one data placement result. Different $IN$ will conduct different data placement, the key of our novel data placement strategy is how to find the best $IN$. There are numerous $IN$ in a data tree, as shown in Fig. 3, $\{b, c\}$ is a possible INode set, and also $\{b, f, g\}$ and $\{d, e, c\}$, INode could be leaf node, e.g., $\{b, l, m, g\}$, $\{b, f, n, o\}$, $\{b, l, m, n, o\}$, etc. The extreme case of $IN$ is all $IN$odes are leaf nodes such as $\{h, i, j, k, l, m, n, o\}$.

How to choose the best $IN$ is obvious a NP-Complete
problem, as discussed in [9] which search the best data placement strategy in parallel DBMSs.

![Diagram](Figure 3. Example for INode)

### 2.2 Data Placement based on INode

The basic idea of data placement based on INode is described below. Given an XML data tree $XT$ and an INode set $IN$, the ancestors of all INodes are $DN$, and the remaining nodes in $XT$ are $TN$. We first duplicate the root tree formed by $DN$ in all sites, then we group the INodes in $IN$ according to their parent node, in each group, we distribute the sub-trees rooted with INodes to different sites with the goal of workload average. On each sites, combine all the sub-trees and $DN$ to form one local data tree.

Our goal is high intra query parallelism, thereby low system response time and high throughput. However, if there are many complex query expressions, there is no any data placement which can make each query having equal workload in all sites. Moreover, we cannot analyze all possible data placement results to find the best, which is obviously impractical. We exploit two heuristic rules to reduce the search space which are discussed in Section 3.

### 2.3 Workload Cost Model

In workload-aware data placement strategies, workload estimation is a crucial part since the outcome of workload estimation will directly affect the data placement result, thereby influence the performance of a parallel system. In this section we propose a cost model to estimate the workload of XML data tree in a shared-nothing parallel database system. These formulae are based not only on the inherent structure of XML data and query expressions but also on the index structures and query methods. Different from the path indices for regular path expressions proposed in [4], we only employ basic indices implemented in XBase [7]. As for some core problems such as ancestor-descendant query and twig join, we implement them with query rewriting and hash join technology though they have been efficiently solved in [1,5]. The reason is that we aim at proposing a workload estimation model to meet demand for common XML query processing techniques instead of certain algorithms or specific systems.

Now we will give our workload cost model to evaluate the workload of a data tree with root $a$. Table 1 gives the variables used in this paper.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_a$</td>
<td>the number of element $a$</td>
</tr>
<tr>
<td>$N$</td>
<td>the number of sites</td>
</tr>
<tr>
<td>$n_{a\circ b}$</td>
<td>the result number of $a \circ b$</td>
</tr>
<tr>
<td>$S_a$</td>
<td>the size of element $a$</td>
</tr>
<tr>
<td>$Q_i$</td>
<td>the $i$-th query in statistics</td>
</tr>
<tr>
<td>$f_i$</td>
<td>query frequency of $Q_i$</td>
</tr>
<tr>
<td>$t_{I/O}$</td>
<td>avg. disk access time</td>
</tr>
<tr>
<td>$v_{net}$</td>
<td>net speed</td>
</tr>
<tr>
<td>$W_a$</td>
<td>workload of tree rooted at $a$</td>
</tr>
<tr>
<td>$W_{label(a,b)}$</td>
<td>workload of label $a$ to $b$</td>
</tr>
<tr>
<td>$f_{label(a,b)}$</td>
<td>frequency of label $a$ to $b$</td>
</tr>
</tbody>
</table>

Table 1. Variables used in this paper

We define the workload of a data tree with root $a$ as:

$$W_a = \begin{cases} 0 & \text{if } a \text{ is leaf} \\ \sum_{b \in \text{child}(a)} (W_b + W_{label(a,b)}) & \text{if } a \text{ is not leaf} \end{cases}$$

where $W_{label(a,b)}$ is the workload of label from $a$ to $b$.

In an XML data tree, the label workload from element $a$ to $b$ is:

$$W_{label(a,b)} = (\frac{PreSize + TempSize}{S_{page}}) \cdot t_{I/O}$$

$$= + (\frac{TempSize}{v_{net}}) \cdot f_{label(a,b)}$$

The first item in the bracket is the cost of disk I/O, including the preprocessing course of hash table and storage of temporary results. The second is the communication cost for transferring temporary results. Consider the cpu cost is in a higher order of magnitude than disk I/O and net communication, we ignore it here. And $f_{label(a,b)}$ is the number of label $(a,b)$.

The PreSize of constructing and using hash table is:

$$PreSize = (3n_a + 3n_b) \cdot S$$

it shows the size of constructing and using hash table. $a$ and $b$ has the same size in our OODBMS, so we use $S$ to represent the object size of $a$ and $b$ when clear.

The temporary results size TempSize is defined as:

$$TempSize = n_a \cdot S_a \cdot (S_a + S_b)$$

it shows the size of temporary results of $a \circ b$. Here operation “$\circ$” mainly refers to parent-child join “/”.

The total number of $label(a,b)$ is defined as:
\[
    f_{\text{label}(a,b)} = \sum_{i=1}^{n} \left( (Q_i \triangle \text{label}(a,b)) \cdot f_i \right)
\]

where \( Q_i \triangle \text{label}(a,b) \) represents the existing number of \( \text{label}(a,b) \) in query \( Q_i \).

Selectivity \( \delta_{a \bowtie b} \) is defined as:
\[
    \delta_{a \bowtie b} = n_{a \bowtie b}/n_b
\]

Selectivity represents the percent of \( n_{a \bowtie b} \) with \( b \) after the operation \( \bowtie \).

3 WIN Data Placement Strategy

In this section we present the framework of our workload-aware intermediary node (WIN) data placement strategy. We first give the holistic algorithm of WIN, and then we will discuss each part in detail one by one. The following shows the comprehensive description of how to find the sub-optimal IN and how to get the data distribution. We consider only element nodes in the WIN algorithm because attribute nodes and text nodes are partitioned according to their parent element nodes.

The first algorithm outlines WIN algorithm for fragmenting an XML data tree into partitions. In this algorithm, a parameter \( XT \) represents an XML data tree to be partitioned, a parameter \( N \) represents the number of sites, \( \text{interList} \) represents the list set of INode and \( \text{finalIN} \) the result of IN. Briefly, the algorithm creates each data placement as follows:
\[
    \text{Win}(XT, N)
\]

1. \( \text{interList} \leftarrow \{\text{root}\} \)
2. \( \text{finalIN} \leftarrow \text{NIL} \)
3. while \( \text{interList} \neq \text{NIL} \)
4. do \( \text{IN} \leftarrow \text{interList}.\text{popFront}() \)
5. if \( \text{Benefit}(\text{IN}, \text{finalIN}) \)
6. then \( \text{finalIN} \leftarrow \text{IN} \)
7. \( \text{Expand}(\text{interList}, \text{IN}) \)
8. return \( \text{Partition}(\text{finalIN}) \)

Next procedure outlines \( \text{Partition}() \) for distributing an data tree with a given INode set \( \text{IN} \) to different sites. With the view to get high intra query parallelism, combining with the special structure of XML document and the characters of query expressions, we first group nodes in \( \text{IN} \) with the same parent with the goal that the sub-trees in each group have the similar structure, and then partition sub-trees in each group to different sites with workload balancing to guarantee high intra query parallelism. After this procedure, each site has a local XML data tree.
\[
    \text{Partition}(\text{IN})
\]

1. \( \text{DN} \leftarrow \bigcup_{E \in \text{IN}} \text{Ancestor}(E) \)

duplicate each node in \( \text{DN} \) to all sites \( P_1 \) to \( P_N \)

4. for each element \( E \in \text{IN} \)
5. do group \( E \) according to \( \text{getParent}(E) \)
6. for each group \( G \)
7. do num \leftarrow 1
8. for each element \( E \in G \)
9. do \( \text{workload}_{AVG} \leftarrow W(\text{getParent}(E))/N \)
10. \( \text{workload}_{ADD} \leftarrow 0 \)
11. if \( \text{workload}_{ADD} < \text{workload}_{AVG} \)
12. then \( \text{workload}_{ADD} += W_E \)
13. else \( \text{workload}_{ADD} \leftarrow 0 \)
14. num \leftarrow num + 1
15. return distribution of \( \text{IN} \)

The following procedure describes the \( \text{EXPAND}() \) procedure which expands \( \text{interList} \). Each time we only expand an INode set \( \text{IN} \) in \( \text{interList} \), the total expand number is \( |\text{IN}| \) (\( |\text{IN}| \) represents the node number of \( \text{IN} \)). Each time we pop a new node \( in \in \text{IN} \) and add all its children to \( \text{IN} \) for expanding, this procedure only expands those results which could take more benefit in \( \text{IN} \) and add them to \( \text{interList} \). Suppose, for example, that a node has numerous children, e.g., 10000, the expansion of this node will definitely take a time-consuming operation, and further expansion is unthinkable. As shown in Fig. 4(a), the parent node \( p \) has many children nodes \( c \), if we expand \( p \), then the \( \text{interList} \) will grow huge. Under this case, we add the virtual node called \( \text{VNode} \) to solve this problem. \( \text{VNode} \) is the node between \( p \) and its children \( c \). The gray nodes in Fig. 4(b) are \( \text{VNode}s \). With \( \text{VNode} \) we could solve above problem. We do not store \( \text{VNode} \) while just import it to decrease the computation.

**Figure 4. An Example of VNode**

\[
\text{EXPAND}(\text{interList}, \text{IN})
\]

1. for each node \( in \in \text{IN} \)
2. do \( \text{IN}' \leftarrow \text{IN} \ominus \text{in}.\text{getChildren}() \)
3. if \( \text{Benefit}(\text{IN}', \text{IN}) \)
4. then \( \text{interList}.\text{pushBack}(\text{IN}') \)
5. return \( \text{interList} \)

Search all possible chances to find the best result is an NP-complete problem, so we add two heuristic rules in computing whether new INode set \( \text{IN}_1 \) can take more benefit than the old INode set \( \text{IN}_2 \) and judge to push it or not, with which to reduce the search space. It is shown in the procedure \( \text{Benefit}() \). From
The extreme case is full duplication. Numerous studies have shown that full duplication leads to reduced performance and this has lead many to believe that partial duplication is the correct solution. Therefore we exploit two heuristic rules to simplify search space and avoid full duplication. They are shown below:

1. If $IN_1$ takes worse workload distribution than $IN_2$, similar to hill climbing method, we assume that there is no meaning to continue expanding and we do not expand in this case;
2. Even if $IN_1$ takes better workload distribution than $IN_2$, the duplication cost is higher than the benefit it takes; we also do not expand in this case.

**Benefit($IN_1$, $IN_2$)**

1. **Partition($IN_1$)**
2. $w_1, w_2, . . . , w_N$ is $IN_1$’s workload from $P_1$ to $P_N$
3. **Partition($IN_2$)**
4. $w_1', w_2', . . . , w'_N$ is $IN_2$’s workload from $P_1$ to $P_N$
5. $W_{IN_1} ← \sum_{i=1}^{N} w_i$
6. $W_{IN_2} ← \sum_{i=1}^{N} w'_i$
7. $Avg_1 ← W_{IN_1}/N$, $Avg_2 ← W_{IN_2}/N$
8. $Max_1 ← \max(w_1, w_2, . . . , w_N)$
9. $Max_2 ← \max(w_1', w_2', . . . , w'_N)$
10. $Ben_1 ← Max_1 – Avg_1$, $Ben_2 ← Max_2 – Avg_2$
11. if $Ben_1 \geq Ben_2$
12. then return true
13. else $\Delta_{ben} ← Ben_2 – Ben_1$
14. $\Delta_{dup} ← (W_{IN_1} – W_{IN_2})/N$
15. return $(\Delta_{ben} – \Delta_{dup}) > 0$

### 4 Performance Evaluation

$WIN$ and PSPIB are both based on the idea to partition path instances to form sub-trees, $WIN$ is aim at workload balance and the goal of PSPIB is the average of path instances. To compare these two data placement strategies, therefore, we implemented them in a native XML database and completed performance test. In this section, we present an experimental evaluation of our novel workload-aware data placement strategy $WIN$ and PSPIB. We start with a description of our experiment platform and test data set in Section 4.1. Section 4.2 presents a result evaluation and performance evaluation of our algorithms.

#### 4.1 Experiment Platform and Test Data

The experiment platform used is a Network of Personal Computers–NOPC. There are six nodes; each node has a PIII 800HZ CPU, 128MB of memory and a 20G hard disk. The operating system used is Sun Unix(Solaris8.0) Intel platform edition of Sun Microsystems. They form a shared-nothing parallel XML database system that communicates through 100Mb/s high-speed LAN. One of them acts as scheduler and the others as servers. We employ a distributed and parallel OODBMS FISH [12] to store XML data. The testing programs were written in Forte C++ and INADA conformed to ODMG C++ binding. Because of establishment limitation, we can only use six nodes to test now, however, our algorithm could be easily extended to more nodes and our current experiment could basically show the performance trend. And we will give the test of more nodes in the future.

We adopt XML Benchmark Project [10] proposed by German CWI as our experiment data set. The XMark data set provides a large single scalable document; each is modelled after the same DTD given in [10]. The test data consists of 20M, 40M, 60M, 80M and 100M single XML document. We use a comprehensive query set of 20 queries [10] to test the performance of these two data placement strategies.

#### 4.2 Experimental Results and Performance Evaluation

$WIN$ vs PSPIB: In this section, we evaluate the performance of $WIN$’ strategy and compare it with PSPIB. To assure the impartial performance evaluation of these two data placement, we use the same query method for these two strategies. With the index structures PNameIndex, NameIndex, TextValueIndex and AttributeValueIndex proposed in [8], we adopt query rewriting and hash join operation for all query expressions. To make experimental results more accurate, we executed each query expression 5 times, the first time is cold results and the latter is hot results. The average time of the hot results is the final response time. Since there are 5 test documents from 20M to 100M, each is tested from 1 site to 5 sites for all queries, the test work and experimental results are very large. We evaluate two key properties in measuring a parallel database management system: speedup and scaleup. In view of the memory limitation, when handling huge XML documents such as 80M and 100M, one site is always the exception point in performance evaluation. In speedup performance evaluation, therefore, we begin with the case of two sites. Not all performance curves are given.
here because of the space limitation.

We first give the performance curves of $Q_5$, this query is to test the casting ability. Strings are the generic data type in XML documents [10]. Queries that interpret strings will often need to cast strings to another data type that carries more semantics. The query requirement is shown below:

*How many sold items cost more than 40?*

This query challenges the DBMS in terms of the casting primitives it provides. Especially, if there is no additional schema information or just a DTD at hand, casts are likely to occur frequently.

The speedup curves of $Q_5$ under two partition strategies were shown in Fig.5–Fig.9. It is easy to see that in this case both strategies have good speedup performance. It demonstrates that our native XML database has good support for casting primitives.

Both $WIN$ and PSPIB’s scaleup curves of $Q_5$ were shown in Fig.10. One site tests 20M document, two sites 40M document, three sites 60M document, four sites 80M document and five sites 100M document. From it, we can clearly see that: with the increasing of the site number, the scaleup curve of $WIN$ decreases smoothly and is close to linear. On the contrary, with the going up of the site number, the scaleup curve of PSPIB changes irregularly and embodies a vibrate change trend. So we can get the conclusion: in this query model, $WIN$ has better scalability than PSPIB.

Next we give the performance analysis of $Q_{17}$. This is to test how well the query processors knows to deal with the semi-structured aspect of XML data, especially elements that are declared optional in the DTD.

The query requirement is shown below:

*Which persons don’t have a homepage?*
speedup but WIN is more stable. With the increasing of document size, nevertheless, it is obviously that the speedup of WIN is better than PSPIB. The reason is that this query contains twig query. There are some efficient twig join methods [3, 5, 6] in single processor enviroment. This query is decomposed into several sub-queries through query rewriting, when the documents are small, the communication cost is low and only a nice distinction in speedup, while with the increasing size of documents, PSPIB has large communication cost and WIN has little communication cost. Thus WIN has better speedup performance than PSPIB in this case.

Both WIN and PSPIB’s scaleup curves of $Q_{17}$ were shown in Fig.16. From it, we can clearly see that: with the increasing of the site number, the scaleup curve of PSPIB changes irregularly. As to the scaleup of WIN, the case of three processors is a exception point, WIN has better tendency except this point. Therefore, WIN has better scaleup performance in this case.

Besides above query expressions, the query set includes many complex query expressions which include multiple query paths. Finally we give the performance analysis of $Q_{20}$. This query is about aggregation. The following query computes a simple aggregation by assigning each person to a category. Note that the aggregation is truly semi-structured as it also includes those persons for whom the relevant data is not available. It is shown below:

\[ \text{aggregate} \]

5 Conclusions and Future Work

In this paper, we proposed the concept of INode and developed a novel workload-aware data placement strategy, WIN, which partitions XML data tree in parallel XML database systems. Extensive experimental results show that our WIN strategy has much better speedup and scaleup performance than previous data placement strategy PSPIB which is based on XML inherent structure.

As for future work, we plan to further investigate workload-aware data placement strategy and exploit different strategies for different requirements. There are also many challenges in expanding this strategy about replacement problem when the XML document is updated or query set is changed.

Acknowledgement

This work is partially supported by the CUHK strategic grant(numbered 4410001).

References


