DataGuide-based Distribution for XML Documents

© Alexander Kalinin

Institute for System Programming of the Russian Academy of Sciences
allex.kalinin@gmail.com

Abstract

Distribution is a well-known solution to increase performance and provide load balancing in case you need optimal resource utilization. Together with replication it also allows improved reliability, accessibility and fault-tolerance. However since the amount of data is large there is a problem of maintaining meta-information about distribution and finding needed data fragments during execution of queries. These problems are well understood but they have not received much attention in the context of XML data management. This paper presents research-in-progress, which examines the possibility of management of meta-information about XML data distribution extending auxillary index structure called DataGuide.

1 Introduction

Distribution and replication are often used in data management to provide load-balancing and improve reliability and accessibility. Distribution means partitioning data into some fragments and allocating the corresponding fragments on some number of sites. Replication deals with problem of allocating the same fragments on different sites. Since these terms are often used in the same context, in this paper we will be using terms "replication" and "distribution" interchangeably. This is justified since we assume distribution of data into fragments and replication some of the fragments on multiple sites.

When using replication numerous problems arise. Among them we first consider:

1. Determining fragments to distribute and replicate and their placement.
2. Management of meta-information about distribution.
3. Management of corresponding fragments and replicas, e.g. consistency of replicas during updates.

These problems have been well researched in the relational world but to the best of our knowledge they have not received much attention in the context of XML data management. However there are exist native XML databases ([1] or [2] for example), which deal with large amount of XML data. For them data replication could give the same benefits as for relational databases. Let us consider a small example:

Proceedings of the Spring Young Researcher's Colloquium on Database and Information Systems, Saint-Petersburg, Russia, 2009

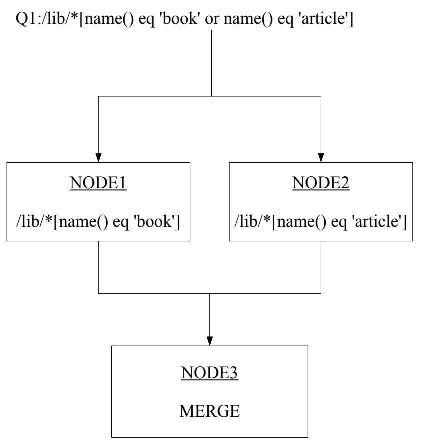

Q1:/lib/*[name() eq 'book' or name() eq 'article']
NODE1
/lib/*[name() eq 'book']
NODE2
/lib/*[name() eq 'article']
NODE3
MERGE

Figure 1: Example of load balancing

Example 1. *Let us look at Figure 1. This figure illustrates simple load balancing example. There is a query $Q1$, which retrieves book and $article$ elements. If book elements were distributed to $Node1$ and $article$ elements to $Node2$ then this task could be done in parallel on these nodes. Then result would be merged on some $Node3$, which could be an original node query had arrived to. Without such distribution $Node3$ would process such query by itself and probably sequentially by retrieving $book$ nodes and then $article$ nodes.*

Now let us review three aforementioned problems in the context of XML data management.

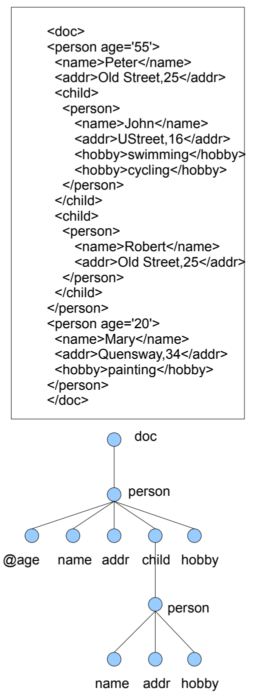
First of all, there is a problem of determining distribution fragments and their placement. This implies choosing fragment's size. For XML documents two choices seem reasonable: the whole document (document-level replication) or individual nodes (node-level replication). Document-level distribution is a well-known solution in case of small documents or mostly read-only environment. The main benefit here is as DBMS executes some query on document D it can choose any of the sites the document D resides on to execute the query. This, at the same time, provides good load balancing and resource utilization during read-only queries. However there is a problem of managing such replicated documents when updates arrive. Since document D is replicated fully among some sites update of any of its part propagates to all of these sites. In case such updates are common they can destroy all benefits we receive from such replication scheme. Another problem here is the size of the fragment. In case of large documents such replication becomes quite space-inefficient. With regard to this we believe that replication based on nodes is a more efficient solution in general, when database may contain documents large in size or when updates are not uncommon. So in this paper we imply node-level replication.

If we use node-level replication we must deal with large amount of meta-information about distribution, e.g. on what sites particular nodes reside. This task in not trivial since the number of nodes in a large XML document may be quite high. The efficient management of such data is the main theme of the presented research. The most common way to deal with such information is to use some kind of auxillary index structures. In this paper we propose using DataGuide[10] as such index. We will describe DataGuide in details and give some examples in the next section. Here it is sufficient to say that DataGuide resembles path-index in such way as every possible path in a document is represented in a corresponding DataGuide structure. Since nodes are located by paths in every XML document we believe this structure to be the most appropriate solution in case of node-level replication. Moreover DataGuide is used in some native XML implementations (e.g. Sedna[2]) for optimization of XPath[3], which is a navigational language for XML documents and also deals with nodes. In such systems using DataGuide to store replication schema means integration with query executor straightforward. Such sound integration allows to receive most benefits from replication since one of its main goals is to make query evaluation more efficient.

At first we propose straightforward and most obvious way to extend DataGuide, which requires minimal modifications. Such DataGuide is then replicated itself among participating sites. Then, the more elaborate approach follows, which includes extending DataGuide, distributing it in some fragments and replicating them only for nodes that really need it. Such approach creates multi-level environment, hence the name $MLDG$ (Multi-Level Data Guide).

Last problem remaining – maintenance of corresponding replicas. The most infamous problem here is to keep replicas in consistent state. Much work has been done in this field outside of XML data management. But since most algorithms deal with abstract "data elements" and do not impose restrictions on underlying data model they can be used for dealing with replication of XML data as well. So we believe it to be a separate problem and do not include any suggestions on the problem in this paper.

The rest of the paper is organized as follows. In the next section we describe DataGuide structure and give some examples of its usability during evaluation of some path-expressions. Then, in Section 3 we discuss the possibility of using DataGuide to store information about replication as well. First we give there the most straightforward approach and then we move further to the more elaborate one, which is our main research proposal. Section 4 gives a brief explanation of work to be done for this research. In Section 5 we describe related work. And Section 6 concludes this paper.



```
<doc>
<person age='55'>
  <name>Peter</name>
  <addr>Old Street,25</addr>
  <child>
    <person>
      <name>John</name>
      <addr>UStreet,16</addr>
      <hobby>swimming</hobby>
      <hobby>cycling</hobby>
    </person>
  </child>
  <child>
    <person>
      <name>Robert</name>
      <addr>Old Street,25</addr>
    </person>
  </child>
</person>
<person age='20'>
  <name>Mary</name>
  <addr>Quensway,34</addr>
  <hobby>painting</hobby>
</person>
</doc>
```

Figure 2: Example of DataGuide

2 DataGuide

DataGuide is a structure that describes different paths of XML document. More precisely:

Definition 1. *DataGuide for XML document D is an XML document DG with following properties:*

1. *For every path in D there is a unique equivalent path in DG*
2. *For every path in DG there is a equivalent path in D*

Figure 2 presents an example of DataGuide. Top half contains document D and the bottom half contains the corresponding DataGuide. One of the most important features of DataGuide is that it effectively serves as a path-level index. Since XPath is the main navigational language for XML documents DataGuide is used in its optimizations. For example, in Sedna XML database nodes of DataGuide point to the corresponding nodes of a document, which allows quick evaluation of XPath queries. Let us look at some example.

Example 2. *Consider query Q1: `/doc/individual`. Query executor would see that such path does not exist and return empty sequence. It does not even have to look into any nodes.*

Now consider Q2: `/doc/person/child`. In this case query executor could find corresponding document nodes by looking at DataGuide and using it as an index.

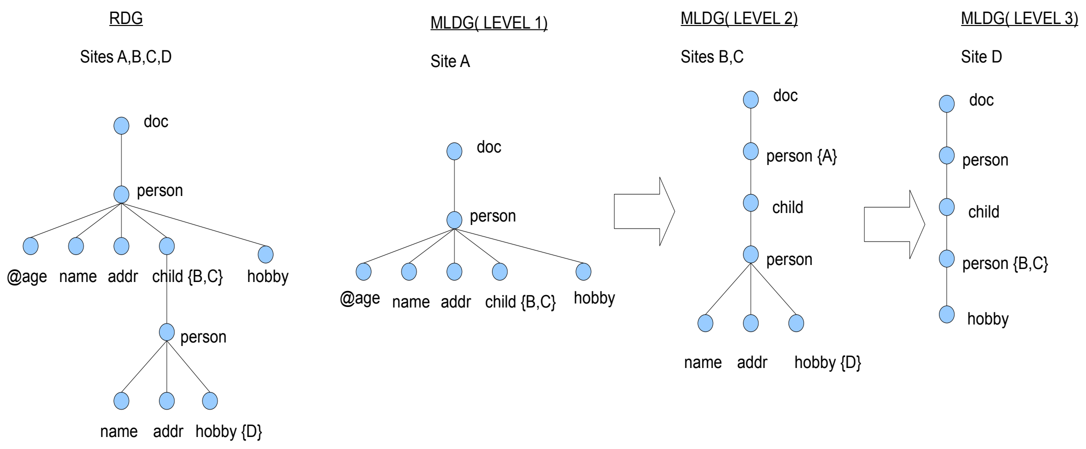


Figure 3: Multi-level DataGuide

In fact, in this case it would not look at any unnecessary nodes at all.

At last, consider `Q3: /doc/person[./child]`*. There is a path* `/doc/person/child` *in DG but according to the definition it does not have to be unique. This means that not every "/doc/person" node has "child" node as a child. However in this case DataGuide allows efficient iteration over "/doc/person/child" nodes and obtaining their parents, which would be resulting "/doc/person" nodes.*

This example shows that DataGuide can be used as effective XML data map. We believe it is straightforward to extend it to be able to support information about distribution, i.e. serve as a distribution map. Since we consider node-level replication here and nodes are accessed by path expressions DataGuide is a natural choice. Moreover it would allow easy integration with query executors, which are aware of path expressions anyway. In the next section we will present extensions to DataGuide to support replication.

3 Extending DataGuide with replication

In this section we discuss the possibility of tuning DataGuide to support replication. We use the term "site" to distinguish nodes on which data reside from XML nodes.

First and the most straightforward way is to extend DataGuide with information about sites the particular nodes reside on. Consider DataGuide DG from Figure 2. Let us assume that `/doc/person/child` nodes are replicated among sites B and C, and site A contains all other nodes except `/doc/person/child`. Then we can store such information in the nodes of the DataGuide. For example, node `/doc/person/child` would contain pair (B, C) allowing to find sites corresponding document nodes reside on. It is a matter of choice whether to propagate such information to the children of `/doc/person/child`. We could assume that without any explicit directions node resides on the same site as its nearest ancestor. In this case we would write pointer to site A only for node `/doc`. We will call such extended DataGuide RDG (Replication-aware DataGuide).

The proposed scheme allows executor to easily redirect part of a query to the specified sites. For example, for query `Q3: /doc/person/child[@age='15']` we would go to one of the nodes B or C and continue query evaluation there. For DataGuide-aware executor traversing DataGuide would be part of the job anyway. Other executors could evaluate $Q3$ on RDG since it is structured as XML document. The main caveat here is that in order to work efficiently such RDG should be fully replicated between all sites we use to answer queries. The main benefit here is that every site knows about data allocation and can redirect parts of a query right to the corresponding nodes. Moreover it can answer some queries locally without even accessing the other machines. As an example consider query $Q4 : fn : count(/doc/person/child/brother)$ received by the site A. Since the corresponding path is absent from RDG then A can give the answer, 0, right away even though it does not contain `/doc/person/child` nodes itself.

Despite of some benefits the RDG approach also has some drawbacks. Since we must replicate RDG to every site it becomes somewhat hard to maintain it in case of updates. In fact updates can be of two types. First ones are updates of corresponding XML document. For example, consider adding node with the name "child" to some of the `/doc/person/child/person` nodes. Since the corresponding path is absent from the DataGuide we would need to update it for every site. So such simple insert of a node becomes an update of the entire cluster of sites. It would be beneficial to update just sites B and C where `/doc/person/child/person` reside. The second type of updates are updates of meta-information. For example, if we want to replicate `/doc/person/child` to some site D. Again since RDG is fully replicated we should propagate such update to every site. However the last problem could be easily alleviated. For example, for site A we could store information only about site B and site B knows that its data replicated also at site C. In this case this last update would touch only sites B and C.

Considering aforementioned shortcomings we propose another approach. The main idea is to distribute RDG itself creating multi-level environment. We call such extended DataGuide $MLDG$ (Multi-Level DataGuide). Let us look at Figure 3. The left part presents RDG discussed earlier. Nodes `/doc/person/child` are replicated on sites B and C and nodes `/doc/person/child/person/hobby` are placed on D. Since site A does not store all these nodes we can prune RDG tree leaving information about nodes that actually reside here and information on where to find `/doc/person/child`. Such pruned RDG would be replicated on every site that stores replicas of A. Such sites form level one in our distributed environment. Sites B and C receive another variation of RDG. They are aware of the internal structure of `/doc/person/child`, so they contain more information about corresponding subtrees. For example, they know about `/doc/person/child/person` nodes of which A is not aware of. B and C form the second level. Lastly, `/doc/person/child/person/hobby` nodes are distributed on site D. Again, B and C know where to find these nodes, but they are not aware of their internal structure. D receives another modification of the initial RDG as specified on Figure 3.

The main rule here is that every site is aware of the internal structure of its replica. This means that it stores the whole DataGuide only for subtrees belonging to the nodes that are replicated to it. Such DataGuides exclude any subtrees for nodes that belong elsewhere as, for example, the case with `/doc/person/child` on site A. As can be seen on Figure 3 every site on level greater than one also stores ancestor context for its replicas, which is a path from document root to the replicated node. These paths become parts of site's DataGuide too. This allows two important things. First of all, this part of the DataGuide allows pointing back to the previous level. For example, `/doc/person` stores information about A and `/doc/person/child/person` stores information about B and C. Without such "pointers" it would be impossible for such sites to receive queries since they would not know where to redirect them in case they have not got the needed nodes. Another reason is that ancestor context allows higher levels to issue proper, without mangling, path-queries to retrieve needed nodes from lower levels. In this case the structure of a query executor for every site would remain the same because of uniformity of distribution information's structure. Moreover it yields more natural description for disjoint node replicas. For example, if some site would hold `/doc/person/name` and `/doc/person/hobby` replicas it would be more natural to hold it together with from-the-root paths and at the same time this allows to evaluate "usual" queries such as `fn:count(/doc/person/name)`.

Let us see how queries could be evaluated in such multi-level environment.

Example 3. *Consider query $Q1 : /doc/person/child/person/name$ arriving at site A. A cannot answer this query but it knows that `/doc/person/child` nodes are allocated at sites B and C. So it redirects this query to either B or C. B and C indeed contain all the data needed for this query. In fact they can execute this query as it first arrived straight to one of them. This means we do not even need to rewrite the query.*

Consider query $Q1$ arriving at site D. Similar to A it knows nothing about nodes in question. But it knows that `/doc/person/child/person` nodes are allocated at sites B and C. So if somebody could give an answer it is one of them. Again it redirects the query to either B or C.

Now consider query $Q2 : /doc/person/name$. If it arrives at site A the answer could be given right away since A knows it. If the query arrives at site B it must be redirected to site A. B could do that since it knows that `/doc/person` allocated at A from its DataGuide. However if the query arrives at site D it could not be redirected right away since D knows only about B and C nodes. In this case it redirects the query at site B(C) and B(C) would redirect it to A.

Multivelel redirection, such as $D \rightarrow B \rightarrow C$ from the above example shows another problem: if query arrives at some level but must be answered by a level much higher or lower than the current there would be a large number of redirections. Such indirection could be alleviated by allowing levels to see more information about other levels. For example, we could add path `/doc/person/child/person/hobby` to the DataGuide at site A allowing A to redirect the corresponding questions without B or C. In this case the environment becomes more centralized and resembles the RDG approach. However the amount of replicated information is still small compared to RDG and at the same time it would allow processing queries without unnecessary redirections. We could say that there would be a some kind of trade-off between more centralized environment with some performance gains and more distributed environments, which is more robust to the updates of DataGuide.

As for the updates let us look at the following example.

Example 4. *First of all, if we need to add some node that is already reflected in the DataGuide the same rule as for queries applies. Such update would be executed in place or redirected to the corresponding site. It is obvious that no update of DataGuide is needed.*

Updates become more interesting when they involve nodes that are new to a DataGuide hence requiring it to be changed. For example, let us assume that we want to add a child with the name "SSN" to the one of the `/doc/person/child/person` nodes. This requires updating DataGuide but only at sites B and C. Compare it with the RDG approach where such update would touch all sites to update fully replicated RDG. That is one of the examples where maintaing $MLDG$ pays off.

Another benefit of $MLDG$ would be further distribution of XML data. For example, let us assume that we want to reallocate nodes `/doc/person/child/person/addr` from B and C to some site F. In this case we would update only sites B and C since it would be enough to locate these nodes. With RDG we would update all sites since every site must know where to find relocated nodes. This example also illustrates that trade-off we were talking about *earlier. If, for example, we want A to be able to locate aforementioned nodes we would update it as well. Again this alleviates indirection but makes changing data allocation map less straightforward.*

This example concludes our proposal. To summarize, the main idea is to use multi-leveled DataGuide ($MLDG$) as a representation of a data allocation map. Multi-levelness allows more efficient updates and evolution of the data allocation schema, i.e. reallocating nodes to another sites.

There is much to be done in this research and in the next section we will give a brief summary of future work.

4 Future Work

In this section we give a brief summary of further directions of our research. First of all, when replicating XML data we must be aware of so-called document order. Imagine, for example, that we want to retrieve some of the `/doc/person/child/person` nodes. Such query would be easily delivered to one of the nodes B and C. But then we must face the fact that some parts of the nodes in question ("hobby" nodes in our example) are located on another site. We must assemble all these parts from sites B (C) and D in the document order. Document order problem does not end with serialization however. Most update extensions to XQuery allow user to specify the exact placement of the inserted node. Again original DataGuide is not suited for the purpose of storing such positional information. Maintaining information about document order is straightforward in the absence of distribution on the internal representation level, but in the distributed environment it could become not-so-trivial task. We believe that using some of the node identifier (NID) schemes can be a life-savior here, but we must elaborate on some effective way to store such information in $MLDG$.

Another direction for the research could be to look more closely on some of the XPath evaluation methods and thinking about easy integration of $MLDG$ with such methods. Such integraton would allow more effective path-query evaluation in the distributed environment and gaining the most benefits from the replication.

And last but not least, we plan to do some experiments to show benefits of the presented approach. There could be a performance decrease because of redirecting queries to another sites, but it should be tolerable in one cases and completely surpassed by benefits of replication in another. As for the updates it should be beneficial to RDG approach for the reasons presented in the previous section.

5 Related Work

There is a large amount of work concerning replication in relational databases(for example [8, 9, 5, 4, 11]). Some of such papers deal with replication of strictly relational data. However since relational and XML data are different in structure these works are not suitable for our purpose. Since XML data could be represented as a tree with arbitrary depth level this implies somewhat more complex map than in the case of "almost flat" (table-oriented) relational data. But other works in this field discuss correctness problems for replicated data and propose algorithms that guarantee one or another level of correctness. XML replication can certainly benefit from these works, but this is not a part of our research. It is more of the third problem we described in "Introduction" since such algorithms usually work without assuming something about underlying data structure.

Now we discuss some works here that deal with distribution of XML data. First one [6] discusses similar problem of representing distribution map. Authors propose $ReplicationGuide(RG)$, which resembles RDG we discussed earlier. However in this paper authors are more concerned with performing structural joins and maintaining information about different physical paths that correspond to one RG path in the form of $\mu PIDs$. This is somewhat similar to maintaing positional information we discussed in the previous section. However, authors do not divulge into details whether $\mu PIDs$ could be used to solve serialization or update problems we mentioned earlier. Moreover this approach lacks in two ways. Firstly, RG is replicated on every site. As we have seen it can be cumbersome in some cases. Secondly, this approach and $\mu PIDs$ particulary are too dependent on DTD or $XMLSchema$, which can be bad in cases data does not follow such rigorous structure.

Second paper [7] discusses partial evaluation problem in a distributed environment. The proposed algorithms deal with so-called "Boolean XPath queries", which is a subset of XPath queries that answer "true" or "false" depending on existence of nodes corresponding to path-expression. Authors propose efficient distribution algorithm, which guarantees some nice properties. However this work is related to the problem of distributed query evaluation and does not discuss any problems of maintaining distribution map.

The final paper [12] deals with interesting problem of caching results of XPath queries in a peer-to-peer environment. Authors propose two approaches: $IndexCache$ and $DataCache$. The former involves storing results on peers that requested them and maintaining prefix-based index to allow other peers to access it. Such prefix-based index is distributed among peers for efficient querying. DataCache involves storing results on particular sites since every site maintains its own portion of query space. This allows to eliminate redundancy. Notice that prefix-based index resembles DataGuide approach as its primary goal is to path-index cached results. So this has little to do with querying distributed database but this technique can be beneficial as a performance increasing add-on.

6 Conclusion

This paper is a research-in-progress discussing some aspects of managing replication for XML data. Replication is an important task for any DBMS natively managing XML data. Managing meta-information about replication, i.e. some kind of replication map is not an easy and trivial task. In this paper we have discussed approaches to manage such information. We have described RDG, extended version of well-known DataGuide, and more elaborate $MLDG$, which is a multi-leveled version of

RDG itself. We have provided numerous examples to show benefits of our approach during evaluation of query and update statements. Also we have discussed directions for future work to continue the presented research.

References

[1] eXist Native XML Database. http://exist.sourceforge.net/.

[2] Sedna Native XML Database. http://modis.ispras.ru/sedna.

[3] XML Path Language (XPath). http://www.w3.org/TR/xpath.

[4] Fuat Akal, Can Türker, Hans-Jörg Schek, Yuri Breitbart, Torsten Grabs, and Lourens Veen. Fine-grained replication and scheduling with freshness and correctness guarantees. In *VLDB*, pages 565–576, 2005.

[5] Yuri Breitbart, Raghavan Komondoor, Rajeev Rastogi, S. Seshadri, and Abraham Silberschatz. Update propagation protocols for replicated databases. In *SIGMOD Conference*, pages 97–108, 1999.

[6] Jan-Marco Bremer and Michael Gertz. On distributing xml repositories. In *WebDB*, pages 73–78, 2003.

[7] Peter Buneman, Gao Cong, Wenfei Fan, and Anastasios Kementsietsidis. Using partial evaluation in distributed query evaluation. In *VLDB*, pages 211–222, 2006.

[8] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with ordering guarantees. In *ICDE*, pages 424–435, 2004.

[9] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *VLDB*, pages 715–726, 2006.

[10] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445, 1997.

[11] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *SIGMOD Conference*, pages 173–182, 1996.

[12] Kostas Lillis and Evaggelia Pitoura. Cooperative xpath caching. In *SIGMOD Conference*, pages 327–338, 2008.