Freshness control of XML documents for query load balancing

Stéphane Gançarski, Cécile Le Pape
LIP6, University P.&M. Curie Paris, France
Firstname.Lastname@lip6.fr

Alda Lopes Gançarski
GET/INT, CNRS UMR SAMOVAR, Evry, France
Alda.Gancarski@int-evry.fr

Abstract

We present an approach for controlling the freshness of replicated XML documents. The main idea is that read-only transactions may accept to read stale data, provided they can express an upper bound on the staleness of the data they read. Controlling the freshness of data accessed by read-only transactions greatly improves load balancing since it allows for choosing a node for executing the transaction even if it is not perfectly fresh. Such a routing is based on detecting which parts of a document are likely to be updated by a given transaction. Due to the rich nature of XML, the problem is quite more complex than for relational SQL data. We present a new algorithm for conflict detection between transactions, needed to estimate freshness of data according to the missing transactions on a node. We also present new freshness measures, in order to take into account the structure/content nature of XML.
Keywords: *Freshness control, load balancing, transaction conflict detection, lazy replication*

1. Introduction

Data replication is a well known solution for improving data availability and reducing query response time through parallelism. This is particularly relevant for XML documents, which are usually handled by read-intensive applications such as Web farms or XML warehouses. We consider hybrid workloads, composed both of update transactions and read-only transactions. Updating transactions are composed of at least one update operation (insert, delete, replace, etc.). Read-only transactions are called *queries* in the remainder. Some projects, such as PowerDB-XML [2], use eager replication: all the copies of a data are updated by the same transaction. This offers strong consistency but at the cost of an overhead that prevents from scaling up and is not well adapted to large scale networks. Other project, such as Xyleme [1], use lazy replication : only one copy is updated by the client transaction, the other ones being synchronized later through subsequent transactions. This is more flexible but raises two issues : global serializability and freshness control. Global serializability, which ensures consistency, implies that updating transactions must be executed on all the nodes in compatible orders. Freshness control allows for a better load balancing of read-only transactions. Indeed, since queries do not modify the database, they may read stale data, *i.e.* data not perfectly fresh, without introducing inconsistency in the document database. However, users would accept to read stale data only if they can control data freshness, *i.e.* provided they can express an upper bound on the staleness of the data they read. Data staleness represents the divergence between the value of a data copy on a node and the value the data would have if all the updates sent to the system would have been applied to the copy. Controlling the freshness of data accessed by read-only transactions greatly improves load balancing since it allows for choosing a node for executing the transaction even if it is not perfectly fresh. This issue has been studied in previous works, but in the context of either relational data, *e.g.* [9], or unstructured data *e.g.* [11]. Recently, we developped Refresco [5, 4], a middleware that controls lazy replication and freshness over a database cluster, each node of the cluster holding a copy of the same relational database. Freshness requirements are defined through a freshness model adapted to relational data. Update transactions are treated as particular case, where data must be per-

[0]Work partially financed by the French ANR Respire Project

fectly fresh. The routing algorithm works as follows. For t an incoming transaction, it detects for each node n the update transactions that conflict with t and have not been executed on n yet. Those update transactions are organized into a dependency graph, denoted *refresh graph*, based on their arrival time and their mutual potential conflicts. If t is an update transaction, it computes the best execution node based on the node load and the propagation cost. If t is a query, some refreshment may not be needed if the query accept to read stale data. Therefore the algorithm prunes the refresh graph until reaching the minimal graph that fulfill the query freshness requirements. Pruning is made according to the quantity of changes made by each update transactions of the refresh graph. Both conflict detecting and pruning use as much static information as possible, obtained by parsing transactions code.

In this paper, we address the problem of controlling the freshness of replicated XML documents, by adapting the strategy and principles of Refresco to XML data. Due to the rich nature of XML, the problem is quite more complex than for relational SQL data. Particularly, two issues must be revisited : conflict detection and freshness measures definition. First, as explained above, conflict detection is at the core of our system, since it is involved in both routing and pruning. In both cases, the issue is to detect conflicts *a priori*, before execution by parsing the transactions codes. Even for SQL expressions, *exact* conflict detection is not always possible using only static information. Parsing the transactions code allows to detect *potential* conflicts by detecting which attribute of which table is potentially read or written. But in general cases, detecting exact conflicts at the tuples granularity comes with the very high cost of reading the database logs. The problem is worse with XML documents which may have recursive and flexible structures. Associated languages takes this into account through specific operators, allowing seeking at an arbitrary depth in an XML tree (such as '//' in XPath) and filtering nodes according to their relative position in the tree. This turns the problem of conflict detection particularly hard, and NP-complete for XPath expressions[8]. However, as in SQL transactions, we only need to detect potential conflicts: our routing and freshness control mechanism is correct provided no conflict is missed. It is however still correct (but less efficient) if it detects "false conflicts". In this case, the only consequence is that it will execute unnecessary transactions on a node to prepare it for an incoming transaction and thus reduce the load balancing quality. Thus, in our algorithm, whenever it is not possible to decide whether there is a conflict or not, a potential conflict is returned. The main issue is then to minimize the number of such cases. Second, freshness measures must be adapted as well. As the XML data model is much richer than the relational model, new measures that take into account structural and textual properties of XML documents must be provided to users, so that they can express freshness requirements relevant to their application.

In order to simplify the problem, we only consider XPath based languages. This simplification is reasonable, as most of the XML languages for updates [3, 10, 6, 7] use XPath to identify the nodes to update, and as XQuery uses XPath expressions to identify nodes/values to retrieve.

The remainder of this paper is as follows. Section 2 presents the algorithm for detecting conflicts between two XPath expressions. Section 3 gives a classification of freshness measures that are relevant for replicated XML documents. Section 4 concludes.

2. Conflict detection

This section presents an algorithm for detecting potential conflicts between two XPath expressions. There is a potential conflict if the intersection of their result is potentially not empty. For example, /a[c]/b and /a/b expressions may concern the same trees, resulting in a potential conflict, while /a/b and /a/c are never in conflict. For sake of readability, the word "conflict" denote a potential conflict in the remainder.

To simplify the problem, the conflict detection algorithm we propose does not cover all the XPath facilities. The simplifications we made are: only abbreviated syntax is recognized; only descendant axis is used; functions are recognized but not interpreted; only a predicate by element is accepted; variables are not accepted; For, If and quantified expressions are not accepted; expressions of type /a(b|c) are not valid, but must be specified as /a/b | /a/c.

2.1 Conflict detection algorithm

Let X and Y be XPath expressions. When at least one of them, let us say X, is composed of different paths (*e.g.* X = /a/b UNION /a/c), there is a conflict if at least one of those paths (*e.g.* /a/c) is in conflict with Y (*e.g.* Y = /a/c). For simple path expressions (*e.g.* /a/b), conflict detection is made by inConflict() function. X.first refers to the first path step of X without "/", if it exists, and X.rest refers to the remaining steps (*e.g.* X=/a/b/c, X.first=a, X.rest=/b/c). Node and Leaf represent, respectively, an element and an attribute. Other node types (processing instructions, text, comments) are not treated here for simplicity.

```
boolean inConflict(XPath_exp X, XPath_exp Y) {
1. IF (X begins with "//" or Y begins with "//") RETURN TRUE;
2. IF (X.first is Node and Y.first is Leaf) RETURN FALSE;
3. IF (X.first and Y.first are both Node or both Leaf) {
     IF (X.first != Y.first) RETURN FALSE;
     IF (DisjointPredicates(X, Y) == TRUE) RETURN FALSE;
     IF (X.rest==NULL or Y.rest==NULL) RETURN TRUE;
     RETURN inConflict(X.rest, Y.rest);
   }
4. IF (X.first == "@*" and Y.first is Leaf) {
     IF (DisjointPredicates(X, Y)==TRUE) RETURN FALSE;
     RETURN TRUE;
   }
5. IF (X.first=="*" and (Y.first=="*" or Y.first is Node)) {
     IF (DisjointPredicates(X, Y) == TRUE) RETURN FALSE;
     IF (X.rest==NULL or Y.rest==NULL) RETURN TRUE;
     RETURN inConflict(X.rest , Y.rest);
   }
6. IF ((X.first=="@*" and Y.first=="Node") or
     (X.first=="*" and Y.first=="Leaf")) RETURN FALSE;
7. RETURN inConflict(Y, X); }
```

The inConflict() function recursively analyzes the path steps of both expressions, starting from the first step. When there is a possible conflict, the function returns true; otherwise, *i.e.* there is no conflict for sure, it returns false. Note that, each time there is a RETURN instruction, the execution terminates: each condition is tested if the preceding tests failed. The different cases (1. to 7.) are described below.

1. If one of the expressions starts by "//", there is no guaranty that there is no conflict, so the result is true. This is the case, for example, of expressions /a/b and /a//c. We can not know if /a//c does not include a path of type /a/b/c, which is in conflict with /a/b.

2. When first steps of X and Y are Node and Leaf, respectively, there is no conflict (*e.g.* X=b/a and Y=@c).

3. When first steps are both Node or Leaf, one of the following four situations may arise.
a) If first steps are different, there is no conflict.
b) If first steps have disjoint predicates, there is no conflict (*e.g.* /a[@c<10] and /a[c>@15]). Function DisjointPredicates() verifies if first steps of two expressions have disjoint predicates, i. e., returns true if both have predicates and these are disjoint, false otherwise[1].
c) If at least one expression is finished (rest==NULL), there is a conflict (*e.g.* X = /a/b and Y = /a).
d) If previous tests fail, the remaining steps of each expression are analyzed.

4. If first steps are "@*" and Leaf, there is no conflict when they have disjoint predicates (*e.g.* X=@*[.<10] and Y=@a[.>10]).

5. To compare "*" with "*" or Node, one among three situations may arise.
a) If they have disjoint predicates (*e.g.* X=*[.<10] and Y=a[.>10]), there is no conflict.
b) If at least one of the expressions is finished, there may be a conflict and the function returns true (*e.g.* X=* and Y=a/c).
c) If both expressions have more steps, the function is recursively called to analyze the remaining steps.

6. There is no conflict if first steps are "@*" and Node or if they are "*" and Leaf.

7. If the preceding cases fail (*e.g.* Y.first=="*" and X.first is Leaf), inConflict(Y, X) tests the symmetrical cases.

2.2 An example of algorithm execution

To illustrate how noConflict() function works by a simple example, let X=/a/f UNION /a[b]/@* and Y=/a[b]/@d. X is a composed expression, so it is necessary to verify that none of the component paths are in conflict with Y. We start by comparing /a/f with Y. Both first steps are Node, corresponding to case 3. These nodes are equal, with DisjointPredicates() returning false (only one of them has a predicate) and expression /a/f having more steps. Consequently, instruction d) of case 3 is executed and the function is applied for the remaining steps: inConflict(/f, /@d). Now, we have a Node and a Leaf. Accordingly with case 2, /a/f and Y do not conflict.

Now, we compare the other component path of X (/a[b]/@*) with Y. The first steps of the path expressions are both Node type, corresponding to case 3. These nodes are equal, have the same predicate and have more steps. Consequently, instruction d) of case 3 is executed calling inConflict(/@*, /@d). This time, first steps are "@*" and Leaf, corresponding to case 4. As they have no predicates, there is a conflict between /a[b]/@* and Y.

Since one of the component paths of X is in conflict with Y, there is a conflict between X and Y.

2.3 Implementation and validation

The conflict algorithm was implemented using Java 1.5.0_06. To verify if XPath expressions are valid, a XPath parser was developed using JavaCC (Java Compiler Compiler) generator. The XPath grammar given to the generator is a simplification of the W3C XQuery

[1] Due to lack of space, DisjointPredicates() is not defined here.

grammar. This grammar was reduced in order to accept only XPath expressions (*e.g.* `SORTBY` expressions were removed). Then, simplifications indicated in Section 2 were made. JavaCC generates Top-Down parsers, so left recursion can not be used. That is why we made the last simplification.

We verified the correcteness of the implemented algorithm for one step paths (*e.g.* `/a` and `/b` return false), multiple step paths (*e.g.* `/a/b/c/*` and `/a/b/c/d` return true), paths including predicates (ex: `/a[p]` and `/a[NOT p]` return false) and composed path expressions (*e.g.* `/a/b UNION /a/c` and `/a/b` return true).

3 Freshness measures

Fresness measures allow users to specify which divergence between replicas is tolerated in order to choose the best replica : the replica which minimizes the response time by minimizing the update propagation cost. This section classifies and describes new freshness measures in a semi-structured context with XML documents. Depending on the application, users choose the adequate measure.

3.1 Freshness definition

Let x be the last version of an XML fragment (specified by an XPath expression) and x' one of its replica. Fragments may be defined at different granularities : as coarse as the whole XML document or as fined as leaves. A freshness measure is a fonction μ such that $\mu(x, x')$ is the divergence between x and x'. Divergence is the updates u_i one must execute on x' in order to bring x' to the same state than x. We denote $U_{x'} = (u_1, u_2, ..., u_k)$ the sequence of updates already applied on x but not yet propagated to x'.

3.2 General measures

This section presents general (not specific to XML replicas) freshness measures.

Boolean measure This measure indicates *if* there is at least one update missing on x'.

$$bool(x, x') = \{ \begin{array}{ll} 0 & \text{if } U_{x'} = \emptyset \\ 1 & \text{if } U_{x'} \neq \emptyset \end{array}$$

Temporal measure This measure indicated the duration since *when* the fragment x' has become obsolete.

$$age(x, x') = dlu(x) - dlu(x'),$$

where $dlu(t)$ is the date of the last update made on a given XML fragment t.

Versioning measure This measure indicates *the number of updates* not yet executed on x'.

$$version(x, x') = Card(U_{x'}),$$

where $Card(U_{x'})$ is the cardinal of $U_{x'}$.

Even if those measures are well-known, they find new usages in XML context. For instance, version measure can be used in append-only RSS feed management to express the number of news missing in the feed.

3.3 Content-based measures

As XML documents usually have textual leaves, we now define freshness measures for the content of nodes. In this section *TC(t)* returns the textual content of the fragment rooted by t.

String_length This measure indicates the *difference of textual length* between x and x'.

$$string\_length(x, x') = \\ |length(TC(x)) - length(TC(x'))|,$$

where *length(a)* returns the length of a.

Chars This measure indicates the *number of characters* that have changed between x and x'.

$$chars(x, x') = Levenshtein(TC(x), TC(x')),$$

where *Levenshtein(a,b)* return the minimal number of char operations needed to transform a to b. For example, *Levenshtein("a kitten", "a sitting")=3*, with the following tranformations :

1. "a kitten" → "a sitten" : substitution of 'k' for 's'
2. "a sitten" → "a sittin" : substitution of 'e' for 'i'
3. "a sittin" → "a sitting" : insert 'g' at the end

Words This measure indicates the *number of words* that have changed between x and x'. This measure can be seen as the same distance than *chars*, with sentence in place of words and words in place of letters.

$$words(x, x') = \\ Levenshtein(s2w(TC(x)), s2w(TC(x'))),$$

where *s2w(s)* is the representation of a sentence s in as a list of words. With the preceding example, *s2w("a kitten")=["a","kitten"]*, *s2w("a sitting")=["a","sitting"]* and *words(x,x')=1* with the substitution of the word "kitten" by "sitting", the word "a" remaining the same.

Numerical measures We consider text nodes with numerical content. Such nodes can be the price of an item, the quantity of an article in a store, the value of an auction, and so on.

$$num(x, x') = value(x) - value(x'),$$

where value(x) is the numerical value of the text node. A variant of this measure is the absolute numerical distance *absnum* .

$$absnum(x, x') = |num(x, x')|$$

Another variant is based on relative distance between x and x'.

$$percent(, x') = |\frac{value(x) - value(x')}{value(x)}|$$

3.4 Tree-based measures

Freshness measures presented above focus on nodes content modification: the structure of the document remain the same, only the value of somes nodes have changed. Since XML documents have a tree structure, we now define tree-based freshness measures. These measures reflect how much a subtree of an XML document have changed. Let x be an XML element node and x' its replica on another node. The subtree rooted by x is denoted *tree(x)*.

Height/Width These measures indicate the difference between the height (respt. width) of $tree(x)$ and the height (respt. width) of $tree(x')$.

$$height(x, x') = |heigth(x) - heigth(x')|$$
$$width_d(x, x') = |width(x, d) - width(x', d)|,$$

where $height(t)$ is the longest path from t to any element node of a given XML subtree $tree(t)$ and $width(t, d)$ is the maximum number of element nodes at a the depth d of a given subtree $tree(t)$. For instance, in a document representing a book composed of chapters, sections and so on, $width_1$ can be used to bound the number of sections inserted or deleted in a given chapter.

Siblings This measure indicates the number of siblings that have changed between x and x'.

$$nb\_siblings_{axis,type}(x, x') =$$
$$|sbl(x, axis, type)) - sbl(x', axis, type)|,$$

where $sbl(elt, axis, type)$ is the number of siblings of an element node elt, for a given $axis$ (*preceding, following, any*) and with a given node type. For instance, $nb\_siblings_{following,"cd"}$ can be used to measure the number of CDs inserted after the last CD bought by the user in a XML compact disc catalogue.

4. Conclusion and future work

This paper presents new contributions for query load balancing over replicated XML documents through freshness control. As far as we know, no other previous work addresses a similar issue. We describe our algorithm for conflict detection, which is the base for transaction routing and freshness computation. We also present a classification of freshness measures suitable for XML data management, with formal definitions. As future work, we plan to formally prove the correctness of our algorithm and extend it to compute the measures defined in Section 3. Then, the algorithm will be implemented and performances evaluated.

References

[1] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic xml documents with distribution and replication. In *ACM SIGMOD International Conference*, pages 527–538, 2003.

[2] T. Grabs and H.-J. Schek. Powerdb-xml: Scalable xml processing with a database cluster. In Blanken, Grabs, Schenkel and Weikum , editor, *Intelligent Search on XML Data*, chapter 13, pages 193–206. Springer, 2003.

[3] A. Laux. XUpdate Working Draft. www.xmldb-org.sourceforge.net/xupdate/xupdate-wd.html, 2000.

[4] C. Le Pape and S. Gançarski. Replica Refresh Strategies in a Database Cluster . In *VECPAR'06 Workshop on High-Performance Data Management in Grid Environments*, Rio de Janeiro, (Brazil), 2006.

[5] C. Le Pape, S. Gançarski, and P. Valduriez. Refresco: Improving Query Performance Through Freshness Control in a Database Cluster. In *International Conference on Cooperative Information Systems (CoopIS)*, pages 174–193, Larnaca (Cyprus), 2004.

[6] P. Lehti. Design and implementation of a data manipulation processor for an XML query language. http://www.lehti.de/beruf/diplomarbeit.pdf, 2001.

[7] P. Poulard. XCL Specification - the XML Control Language. http://disc.inria.fr/perso/philippe.poulard/xml/active-tags/xcl/xcl.html, March 2006.

[8] M. Raghavachari and O. Shmueli. Conflicting xml updates. In *EDBT International Conference*, pages 552–569, 2006.

[9] U. Röhm, K. Böhm, H.-J. Schek, and H. Schuldt. FAS - A Freshness-Sensitive Coordination Middleware for a Cluster of OLAP Components. In *VLDB International Conference*, pages 754–765, Hong Kong (China), 2002.

[10] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. www.cis.upenn.edu/˜zives/research/updatingXML.pdf, May 2001.

[11] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transaction on Computer Systems*, 20(3):239–282, 2002.