Batch Mode Update For View Maintenance Over Semi-structured Data

Dazhi Wang, Junyi Xie
{wangdz, junyi}@cs.duke.edu

Advisor: Dr. Jun Yang
junyang@cs.duke.edu

Abstract

Unlike the structured data organization in traditional relational database management system, the semi-structured data can be irregular and incomplete which is associated with schemas contained in the data itself. The materialized view for semi-structured data needs to be maintained in response to changes of base data. This report addresses the batch-mode updating problem in view maintenance over semi-structured data. We extend the base view maintenance algorithm from [1]. We propose a batch mode update algorithms that does not incur dependence problem when view is updated out of the order of base data updates. Our initial experiments show that when the number of updates is small, overhead of batch mode update as block I/O is on the same level of B+ tree index and hash index. And it takes around two orders of magnitude less access time than these two index structures when number of updates increase to more than 100,000. Additionally, it does not require special index structure if the semi-structured data is stored using relational database which saves the graph structured data by edge.

Introduction

The fact that most data source from web does not conform to traditional relational data model leads to aggressive research work on semi-structured data. Currently most research in this area is aimed at extending database management techniques to semi-structured data. One of these is the incremental view maintenance problem over semistructured data. View over base data can be used to filter the data and restructure it. And views are often materialized to speed up the response time of queries from users when underlying data is remote or response time is crucial. The key problem of view maintenance is that how to keep data in view consistent with the base data. That means when the base data updated, how to update view accordingly. View recomputation from scratch is the simplest way but the most expensive way so it is not realistic to re-compute view in reality. Many viw maintenance techniques have been explored in relational database. The main idea is that by only computing the incremental updates to the view based on the updates to the database, the view maintenance work will become much cheaper than re-computation from scratch. However, unlike the mature incremental view maintenance algorithms for relational database, the incremental view maintenance algorithms for semi-structured data are far from perfect and optimality. There are several reasons. First, the semi-structured data is “schema-less” data, which means it does not have any fixed schema and well-defined key constraints for the stored data. Without schema, the view maintenance algorithms in relational database, which heavily depend on referential integrity do not hold any longer in scenario of semi-structured data. Second, there is no uniform data model for semi-structured data so far. Each model puts some kind of constraint over the data stored, for example, WHAX data model regulates that every edge from a node can be identified by a key value called local identifier. Therefore view maintenance algorithms over one semi-structured data model does not hold on other semi-structured data model in general. And, there is no uniform query language defined for semi-structured data. In contrast to the case of relational database in which SQL is the standard query langue, there are many different and incompatible query languages in semi-structured data, such as XML-QL, Lorel, XQL, XSLT, etc. In 1998, Abiteboul el proposes a simple view specification mechanism and an algorithm for incremental view maintenance over semi-structured data. The data model used is Objective Exchange Model(OEM). However, the algorithms proposed in that paper has some weaknesses that the author left for future work. One of these weaknesses is that it does not support batch-mode view update. As consequence, each update to base data will trigger at lease one view update query evaluation and view is updated in sequence of base update order. In this project, we are aimed at, first to propose an alternative algorithm which supports view update for a set of base data updates, second, it can reduce update cost in batch mode and we discuss the dependence problem to make sure it will not jeopardize the consistency between view and base data since the order of view update is not consistent with order of base data update. The report organizes as following, we first introduce the data model, query language in lorel system. Then we introduce how view is incrementally maintained using the base algorithm introduced in [1]. In next section we introduce the batch-mode update algorithm and discuss the update dependence problem. We then presents the simulation results we have so far to compare the performance between non-batch mode update and batch-mode update. At last, we conclude this report by some observations from this project and some future work.

Base View Maintenance Algorithms

OEM Data Model

In semi-structed data, the database can be described as a labeled, directed graph(OEM, Object Exchange Model) in which each vertex in the graphs represents an object and each object has a unique object identifier (oid). The object can be either atomic object or complex object. OEM is designed to handle the incompleteness as well as the structural and type heterogeneity of data. Figure 1 is a simple example of OEM database, in which semantic information is included in the labels as part of the data and can be exchanged dynamically. In this respect, this OEM graph is self-describing. Figure 1 describes a simple bibliographic database. It is easy to see that there is no schema in database and except the unique object identifier (oid) there is no way to globally identify the objects in graph.

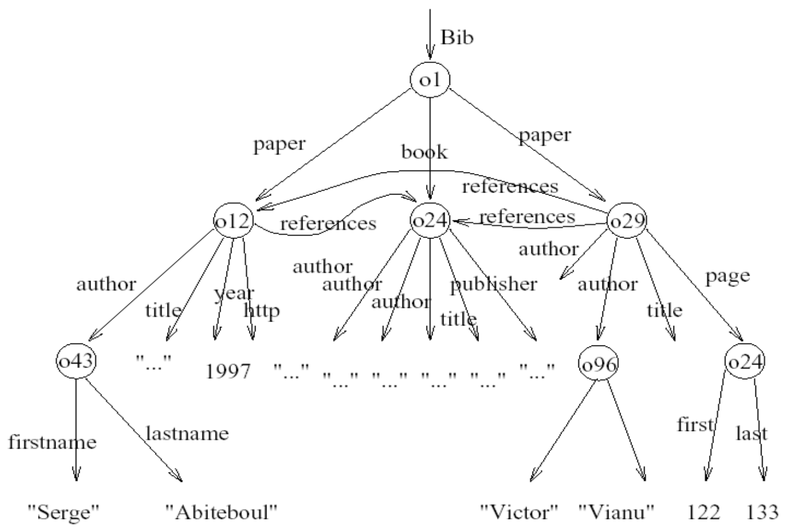

Figure 1 Sample OEM Data Model

Lorel Query Language
The Lorel query language uses the familiar *select-from-where* syntax of SQL and can be considered as extension to OQL that provides powerful path expression for traversing the data and extensive coercion rules for a more forgiving type system. Following example is a query that returns all *Entrée* sub-objects of a resteraunt named "China Hunan" and one of its ingredients has the value "Mushroom"
Exmaple 1
Select *e*
From *Guide.Restaurant r, r.Entrée e*
Where *r.Name = "China Hunan"*
and *e.Ingredient = "Mushroom"*
The path expression in lorel query language is composed of a set of one step path which has the form *x.L y* where *x* and *y* are variables bound to object ids and *L* is variable bound to edge label.

View Definition in Lorel
The view definition in Lorel can import objects and edges from a source database into a view and new objects and edges can be created in the view. Additionally the view specification extends *select-from-where* statement with a *with* clause. Each object and edge along a path in the *with* clause is included in the view. In other words, *select-from-where* can only return a set of objects and *with* clause imports some structure into view.
Example 2 defines a view as a result of example 1.

Exmaple 2
Define view *favoriteEntrees* **as** *Entrees* = **Select** *e*
From *Guide.Restaurant r, r.Entrée e*
Where exists x in *r.Name: x = "China Hunan"*
and exists y in *e.Ingredient: y = "Mushroom"*
With *e.Name n, e.Ingredient i;*

Incremental View Maintenance Algorithms
In brief, the basic view maintenance algorithm can be divided into three steps. First of all, it checks for relevance of update *U* to the view instance *V* defined by the view specification *S*. In this step, an auxiliary data structure named *RelevantOIDs* is used to check whether a base data update is relevant to the view. The *RelevantOIDs* is a set of all object oids which are touched in the course of view evaluation. Any object touched in view evaluation, no matter whether its binding can produce an object into view, is recorded with the variable bound to it. The second step is to generate view maintenance statements. The key idea is to bind the update objects to the view definition and the evaluate it. Since this update is relevant and from set *RelevantOIDs* the variable bound to this object is already known. We can evaluate the statement efficiently than original view specification since some variables in *where* clause are already bound to objects. In the course of evaluating view update statements, deletion should require additional attention. This is because when a view update statement generates an object which should be deleted from view, this object should not be deleted from before checking whether there exists another path in base data that can make this object remain in the view. This extra step does not hold if the base data model is a tree rather than a graph. Example 3 shows how view maintenance statement is generated given an update to base data.
Example 3
Assume an insertion:
<Ins, &10, Ingredient, &15>.
Generated update query:
Select *e*
From *Guide.Restaurant r, r.Entree e*
Where exists x in *r.Name:x = "China Hunan"*
and exists *&15* in *&10.Ingredient: &15 = "Mushroom"*
and *e=&10*
In above example, an edge insertion <Ins, &10, Ingredient, &15> is performed on the base data. Obviously *&10* and *&15* should be bound to variable *e* and *y* respectively in the view definition in example 2. Now we get the view update statement in example 3. It is easy to see the statement in example 3 can be more efficiently evaluated than that in example since it does not need to search the base data graph to bound variable *e* and *y.* The last step is to simply install the objects generated from view update statement evaluation

Weakness of Base Algorithms
The main problem in the above base algorithms we would like to address in this project is that it is not able to support batch-mode view update. Actually, every update to the base data, no matter to what extent it will effect the view, will trigger at least one view

statement evaluation. Next we introduce the batch-mode update algorithms that are able to support batch-mode update to views without tree-like structure generated in WHAX data model. Our main object is to update view for a set of base data updates rather than for single base data update, and we want to see whether it will reduce the update cost in batch mode and we discuss the dependence problems.

Batch View Update without Tree/Graph Structure Generation

In WHAX data model, the view update can be done in batch mode. However, there is an underlying assumption in WHAX model, that is, when update comes to base data, its search path from root to the node updated is already known. Thus multiple updates can be merged together through deep union operator. This is possible in WHAX model since each node in WHAX model can be uniquely identified by the path from root to this node itself. However, it does not hold in OEM model since there is no such presumed key-like path for each update to base data. We can get this path by evaluating this update over and merge paths into a graph structure for all view updates. But this will introduce the same cost of view update in non-batch mode. Therefore, our objective is to do batch-model update without any tree/graph structure generation. Our solution has following two phrases: 1) Classify updates to base data 2) Batch mode view update.

Classify updates to base data

In the first phrase, we classify base updates into different categories. Actually in OEM model, there are two kinds of base updates: edge insertion/deletion and node insertion/deletion. For edge insertion/deletion, we classify updates by the type of edge involved in update. Updates of edge insertion/deletion fall into the same class of updates if the edges involved share the same label. Then we build a table for each class of updates. The schema of this table is *(startOID, endOID, op)*, where *startOID* and *endOID* are the objects of starting and end points of edge, and *op* indicates which kind of operation of updates: insertion or deletion. The value change for label can be seen as a deletion followed by an insertion. For example, we have following set of updates:

Example 4

I(o1, Name, o2), D(o2, Rating, o3), I(o4, Name, o6), I(o1,Entree, o5), I(o7, Name, o9) D(o11, Rating, o12)

We can classify above updates into three categories represented by following tables:

startOID	endOID	op
&o1	&o2	I
&o4	&o6	I
&o7	&o9	I

startOID	endOID	op
&o2	&o3	D
&o11	&o12	D

startOID	endOID	op
&o1	&o5	I

Each table represents a class of updates: *(x.Name y) (x.Rating y) (x.Entree y)*

For node insertion/deletion, actually we can convert it to a set of edge insertion/deletion then classify the node updates as edge updates. Following example shows how to convert a node update into a set of edge updates.

Example 5
Assume node *&o* in base data will be deleted, and there are two edges associated with node *&o*, then when delete *&o,* we convert a node deletion into two edge deletion and one deletion. Obviously, the two edge deletion can be classified into classes in the same way as edge update. Similarly we can convert node insertion into a set of edge insertion and a node update into a set of edge deletion followed by a set of edge insertion.

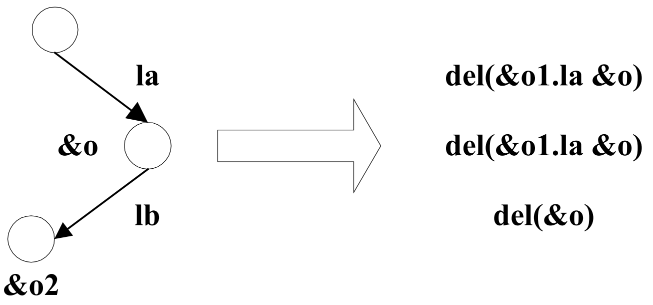

Figure 2 Convert Node Update to Edge Update

Algorithm Outline
We divide the algorithm into two phases. In the first phase, we construct the update table for each view maintenance statement. When an update comes, we first check its relevance to the view definition using the *relevantOID* set. If the *oids* in the update are not in *relevantOID*, we just ignore this update; otherwise we classify the update by its edge label and insert a tuple into the corresponding update table. If this update needs a new maintenance routine to update the view, in this case there is no existing table in which this update can be inserted into, we then create a new table for this kind of updates.

In the second phase, we evaluate each maintenance statement in batch mode. Because we have grouped together the updates belonging to the same maintenance routine, it is possible to batch process the maintenance statement on the whole table, instead of on a per-update basis.

As an example of batch update, assume the semistructured data is represented by OEM model and stored in relational database as an edge table, which has the form <*pid, label, cid*>. For the class of updates which has the form *x.A y*, the view maintenance routine is to first check if *x* has a parent *z* via an edge labeled *B*; then check whether *z* has a child connected by an edge labeled *C*. If yes, we add *z* into the view. To batch process this maintenance statement, we first join the update table corresponding to *x.A y* with the edge table to find all *x*'s parents *z* connected by a *B* edge and create a temporary table , then join this table with the edge table to find all *z* that has an edge labeled *C*, finally add the results into the view.

Dependence Problems
One potential problem for batch update is the ordering of view updates. For the base data, suppose update u_1 is applied before update u_2, but in the batch mode view update, u_2 may

be applied before u_1. If we assume the view update is monotone, there won't be dependence between insertions and insertions or deletions and deletions. But the update dependence problem will arise if one insertion inserts an object into the view, while another deletion will delete the same object in the view. In this case we have to apply the edge insertion and deletion to the view in their original order. We discuss this potential problem under the following 2 cases:

(1) The insertion and deletion involve the same edge. For this case we can simply process it before applying the updates to the view. If an insertion comes, we check the corresponding deletion table to see if a tuple with the same oid and edge label exists. If yes, just remove that tuple. And for an edge deletion, the processing is similar.

(2) The insertion and the deletion involve different edges. In this case the two sets of objects generated by these two updates are distinct. That is, there is no dependence between these 2 updates. The intuition behind this is that we apply the updates to the view after we have already updated the base data, therefore the evaluation of the maintenance statement over the base data always generate the correct result.

Based on the discussion above, the dependence problem in our batch update algorithm can be easily eliminated.

Experiments and Results

To evaluate our batch update algorithm, we make a simulation program that calculates the number of disk I/Os as well as the disk access time. We assume that the OEM model that represents the XML files are stored as edge tables. The database may contain many kinds of indices, but only *Vindex* and *Lindex* are used by the view maintenance statements to get the objects that should be inserted/deleted to/from the XML views.The *Vindex* supports finding all atomic objects with a given incoming edge label and satisfying a given predicate. The *Lindex* supports finding all parents of a given object via an edge with a given label. Since the view maintenance statement has already contained much variable-binding information, it is a reasonable that only *Vindex* and *Lindex* are used to evaluate the maintenance statement. Some of the parameters in our simulated environment are as follows:

Number of update classes	10
Related probability	0.1
Disk block size	4KB
Memory size	4MB
Number of blocks of edge table	6000
Vindex	4 level B+ tree
Lindex	Extensible hash table
Disk seek time	100ms
Disk read time	0.5ms
Max/Min # of simple path expressions	6/2

Table 1 parameters in the simulation

In the table above the related probability represents the percentage of the base data updates that is related to the view definition, the *Vindex* is stored as a 4-level B^+ tree, and the *Lindex* is stored as an extensible hash table. The disk seek time is the time for the head to move to the right block and the disk read time is the time that the head reads one block. The last parameter is the max/min number of simple path expressions in a single view maintenance statement.

In our simulation, we considered 4 methods of evaluating path expressions in a maintenance statement:

(1) Batch mode, which joins the update table with the edge table to get the objects that should be inserted into or deleted from the XML view;
(2) Non-batch mode, this method assumes no index structure exists in the database. For each update, given the variable-binding information, this method will evaluate the path expression by looking through the edge table;
(3) *Lindex* operator, which evaluates path expressions using *Lindex* given the binding information of the child and a labeled edge;
(4) *Vindex* operator, which evaluates path expressions using *Vindex* given the binding information of the parent, a labeled edge and a predicate.

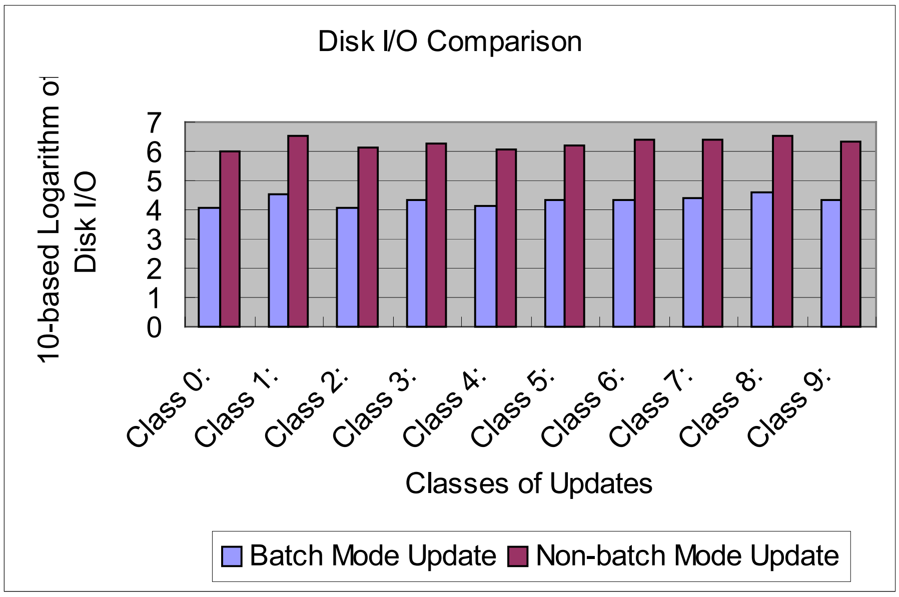

Figure 3 Disk I/O comparison for various classes of updates

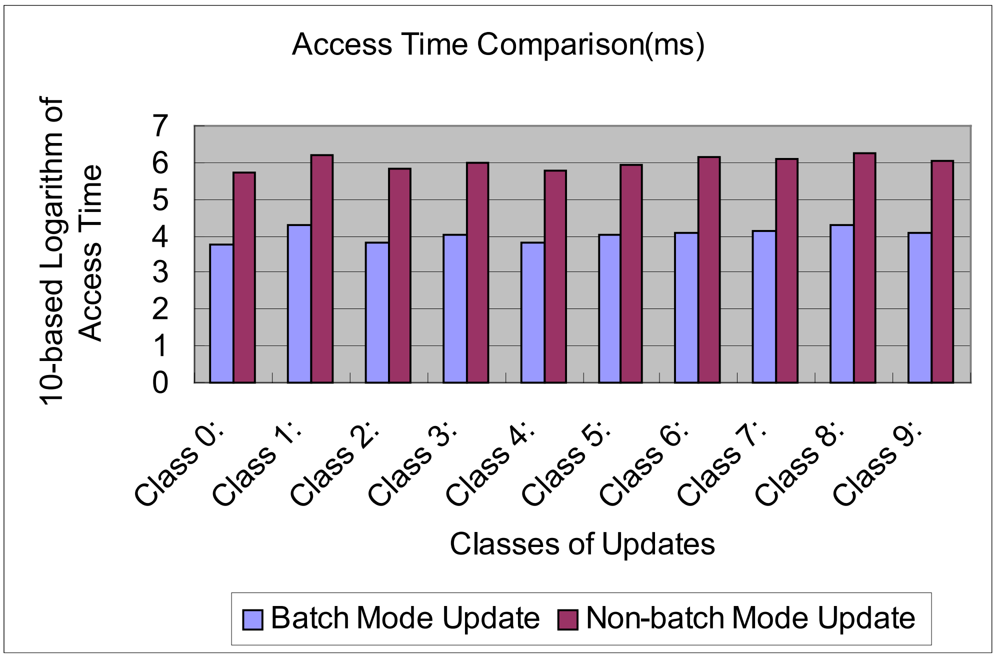

Figure 4 Disk access time comparison for various classes of updates

Figure 5 shows the disk I/O comparison of batch mode update and non-batch mode update for different classes of updates, in which the number of updates is 10000. From this figure we can see that the batch mode update saves a lot more I/O operations, about 2 order of magnitude, than the non-batch mode update. The reason is that for batch mode update, we read the tuples in the update table as much as possible into main memory, and sequentially scan the edge table to join with the update table. In most cases 4MB main memory is enough to hold the entire update table, therefore we only need to scan the edge table once. But for non-batch mode update, the edge table has to be scanned for each single update.

Figure 4 shows the disk access time of batch mode update and non-batch mode update for different classes of updates. We get the similar result as Figure 3. That's because most of the disk I/Os in both batch mode update and non-batch mode update are sequential scans of the edge table, the time to read one disk block is the same for these 2 methods. Therefore the disk access time is proportional to number of disk I/Os.

Figure 5 compares the overall performance of all 4 path evaluation methods: batch mode, non-batch mode, *Lindex* and *Vindex*. When evaluating path expressions using index, some of the path expressions in a maintenance statement are evaluated using *Lindex* and some are evaluated by *Vindex*, but for simplicity in our simulation we assume that the path expressions in a statement are evaluated either all by *Lindex* or all by *Vindex*.

From Figure 5 we can see that when the number of updates is small, the index-based path expression evaluation methods perform better than the batch mode update. However as the number of updates increases, the batch mode update outperforms the index-based methods. The reason is that the index update methods are on a per-tuple basis, when a new update comes, several random disk I/Os are needed to search the index structure, which cost much more time than sequential disk I/Os. Therefore the disk access time for index-based updates grow linearly with the number of updates. On the other hand, for the batch mode update the disk access time will have a remarkable increase only when the main memory can't hold the entire update table. As long as the main memory can hold the update table, the whole edge table only needs to be scaned once, and the disk access time won't change much. What's more, the entire disk I/Os in batch mode update are sequential I/Os, which can save much access time.

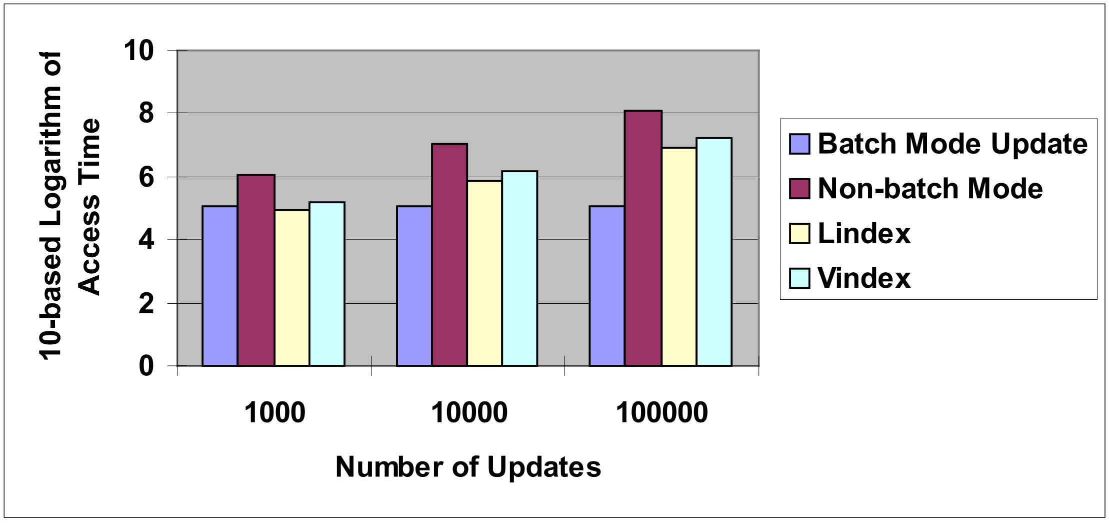

Figure 5 Overall performance comparison

Conclusion and Future Work

In this project we proposed the batch mode update method to update the materialized view of semistructured data. Compared with the approach of recomputing the view from scratch, this method can use the binding information in the view maintenance statement, therefore saving much time compared to executing the view definition query without any binding information; on the other hand, it also saves disk I/Os and disk accessing time by grouping updates that can be processed by the same maintenance statement together into one table, and then evaluating the statement on that table. From our simulation result we can see that when the number of base data updates is large, the batch-mode view update method greatly reduce the disk access time.

For our future work, our idea of grouping updates together only has the potential to do view updates in batch mode, but how to do it really depends on the real systems, for example how the semistructured data is stored, and what kind of index structure it contains, etc. If the semistructured data is stored using a different data structure from edge table, we need to find other ways to do batch updates on that data structure, or design efficient method to construct an edge table from that data structure. And how to cluster the updates to do batch update using some index structures is also an interesting topic.

In our project we only considered the update that has the form <*oid1, label, oid2*>. Because currently there are many ways to store semistructured data, the form for semistructured data update is also different. We need to find ways to group the updates together and batch process them given different update forms.

All simulations so far are based on an assumed environment, to measure the precise performance of batch mode update, we need to test the algorithm using real benchmarks or system simulation.

Reference

[1] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. Wiener. Incremental Maintenance for Materialized Views over Semistructured Data. *Proceedings of the Twenty-Fourth International Conference on Very Large Databases*, New York, August 1998.

[2] J. McHugh and J. Widom. Query Optimization for XML. Proceedings of the Twenty-Fifth International Conference on Very Large Data Bases, pages 315-326, Edinburgh, Scotland, September 1999.

[3] Yannis Papakonstantinou, Vasilis Vassalos, Query Rewriting for Semistructured Data. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Philadelphia, Pennsylvania, June 1999.

[4]Shanmugasundaram, Kiernan, Shekita, Fan, Funderburk: Querying XML views of relational data , VLDB 2001

[5] Yue Zhuge and Hector Garcia-Molina, Self-Maintainability of Graph Structured Views, Technical report, Department of Computer Science, Stanford University, September 1998.

[6] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. Wiener. Incremental Maintenance for Materialized Views over Semistructured Data. *Proceedings of the Twenty-Fourth International Conference on Very Large Databases*, New York, August 1998.