L. Schlesinger, A. Bauer, W. Lehner, G. Ediberidze, M. Gutzmann

Efficiently synchronizing multidimensional schema data

Erstveröffentlichung in / First published in:

CIKM01: International Conference on Information and Knowledge Management, Atlanta 06.11.2011. ACM Digital Library, S. 69–76. ISBN 978-1-58113-437-7

DOI: https://doi.org/10.1145/512236.512246

Final edited form was published in "CIKM01: International Conference on Information and Knowledge Management. Atlanta 2001", S. 69–76, ISBN 978-1-58113-437-7
https://doi.org/10.1145/512236.512246

EFFICIENTLY SYNCHRONIZING MULTIDIMENSIONAL SCHEMA DATA

L. Schlesinger, A. Bauer, W. Lehner, G. Ediberidze, M. Gutzmann
Department of Database Systems, University of Erlangen-Nuremberg, Martensstr. 3, 91058 Erlangen, Germany
{Schlesinger, bauer, lehner, ediberid, gutzmann}@immd6.informatik.uni-erlangen.de

ABSTRACT

Most existing concepts in data warehousing provide a central database system storing gathered raw data and redundantly computed materialized views. While in current system architectures client tools are sending queries to a central data warehouse system and are only used to graphically present the result, the steady rise in power of personal computers and the expansion of network bandwidth makes it possible to store replicated parts of the data warehouse at the client thus saving network bandwidth and utilizing local computing power. Within such a scenario a - potentially mobile - client does not need to be connected to a central server while performing local analyses. Although this scenario seems attractive, several problems arise by introducing such an architecture: For example schema data could be changed or new fact data could be available. This paper is focusing on the first problem and presents ideas on how changed schema data can be detected and efficiently synchronized between client and server exploiting the special needs and requirements of data warehousing.

1 INTRODUCTION

Data warehousing ([12], [13]) has nowadays become a common technology. It has proven itself in everyday life. The success however calls for more extensive use and instant accessibility. In general the goal of a data warehouse is to provide analysts and managers with strategic information about the key figures of the underlying business. For ease of integration data warehouses typically exhibit a centralized architecture ([4], [13]). Usually raw data are collected, purged, integrated and stored within a central database system. The resulting data warehouse database is furthermore enriched with materialized views ([2], [9]) transparently used during runtime to speed up user queries. Sometimes data marts are derived from the data warehouse by storing a part of the data in a separated database system to meet the needs of special applications or user requirements. Based on these databases, OLAP, data mining, or special statistical tools are then used to explore the database. From an architectural point of view, each tool issues queries against the data warehouse, where the answer is computed. The result is sent back to the tool, which displays the result. The most important point is that the query is exclusively computed in the data warehouse, while the tools only help the user to define the query and display the result.

Future Requirements and Appearing Problems

Nowadays a data warehouse is not a subject for specialists. In fact, analyses are done by users in many departments of a company, e.g. purchase, sales or production. Furthermore, users of the data warehouse are residing at different locations or even moving around; for example travelling salesmen or branch offices are not always connected to the central data warehouse but need access to the data warehouse. To meet all these new requirements it appears useful to cache parts of the warehouse at the client side and allow local computation. As storage capacity on the client systems is limited and the users usually need a smaller fraction of the data only a small part of the full data warehouse is stored as a replicate at the client. Assume a sales support system with data about sales values for each customer and subsidiary. A salesman responsible for a certain region is only interested in obtaining the data of his customers and subsidiaries. This information is copied onto his laptop or updated if an old version is already available when connecting to the central warehouse system.

Since from our point of view it is safe to assume that the technical problems like sufficient network bandwidth for synchronization or local computer power and storage capacity are no longer existent in a modern world, the problems of inconsistent data arising from a not permanently connected client to the server are more difficult and needs to be addressed:

- The schema or instance data can be changed on the server. How is it possible to update the information on the client in an easy and efficient way?
- How is it possible to integrate new fact data at the client?
- Which materialized views (summary data, pre-aggregates) should be stored at the client?

This paper emphasizes the first problem. We start diving into more detail by presenting an overview of the overall synchronization process. The remaining sections discuss architectures and algorithms for each step of the synchronization process.

Overview of the Synchronization Process

In general the update or synchronization process within our proposed framework may be subdivided into three steps, two of them are executed at the server while the last step is executed at the client. Figure 1 illustrates the steps in a chronological and local order. Each step is outlined in the following:

- *Server-Update-Phase*
  During the first phase, the server database is updated. Changes of the schema information are imported into the

Final edited form was published in "CIKM01: International Conference on Information and Knowledge Management. Atlanta 2001", S. 69–76, ISBN 978-1-58113-437-7
https://doi.org/10.1145/512236.512246

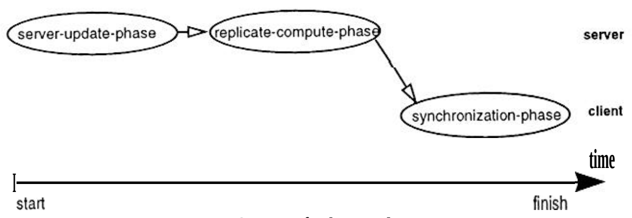

Figure 1: Steps of the update process

database and stored in several specific data structures. Details of this phase, especially the data structures, are explained in section 3 1

- *Replicate-Compute-Phase*
  Replicates are computed during this phase A replicate consists of the changes being relevant to a specific client The detection of changes in the schema of the server database and the composition of the data structure to be transferred to the client during the following step are discussed in detail in section 3 2
- *Synchronization-Phase*
  The last phase consists of the transfer of the replicate computed during the second phase and the update of the local database As this phase is basically not very complicated a further discussion is omitted

The processing of each phase depends on the architecture of the client-sewer-system Therefore, it is possible that the second phase is executed while the client is or is not connected to the server. For this reason it might be necessary to store the replicate at the sewer. Details are discussed in section 2

Related Work

To the best of our knowledge there is not still any paper published dealing with the problem of synchronizing multidimensional schema data between server and client. Indeed ideas from several areas are collected and influences the underlying work

Before discussing the relationship to prior work, we need to introduce the idea of the multidimensional data model. As the model is assumed to he well known we refer to the literature for an in depth discussion ([1], [8], [13], [19],)

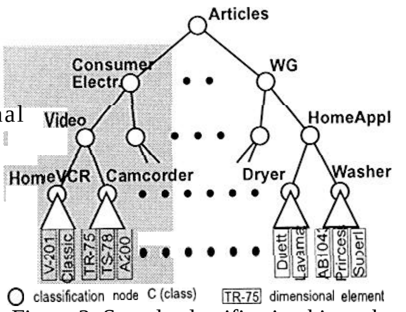

Figure 2: Sample classification hierarchy

The subcubes to be replicated at the clients are defined upon the classification hierarchies of the dimensions specifying the multidimensional data cubes on the server By picking a single classification node of a hierarchy in each dimension as a new root node, a subcube is unequivocally defined. Figure2 shows a sample classification hierarchy of a product dimension. By tagging the node 'Consumer Electronic', a part of the classification tree is selected (grey area in figure 2)

A synchronization of multidimensional schema data is necessary as soon as changes are performed and affect to the specified subcuhe For having a full history of changes the concept of 'versioning' ([3], [17], [18]) is introduced, which forms the basis of detecting and synchronizing changes in the classification hierarchy The history list holds all information about the possible operations on the classification, i.e. deleting or inserting a node or assigning a node to a new parent node Modification of the schema also exhibit an effect to the fact data, which have to he adapted to the new structure. For this problem mechanisms and algorithms are well known from the research area of schema evolution and versioning, where formalisms for updating dimensions are introduced ([10], [1 I])

Besides the multidimensional data model and closely related with the area of versioning we found our discussion on the time stamp idea ([3], [6], [14], [15]). They introduce the idea of marking edges and nodes with numbers representing a logical time

Finally it is annotated that the idea of synchronizing data can be found in [5], where Java objects are synchronized In particular several ideas have strongly influenced the presented architectures given in section 2

Contribution and Organization of the Paper

Detecting changes and efficiently synchronizing data could be a hot topic in future due to the possibilities given by high preforming clients and networks. This paper outlines possible architectures according to the phases of the update process and presents algorithms for processing the synchronization

The organization of this paper is as follows: Section I gives an overview of possible client-server-architectures in a data warehouse environment Necessary data structures and algorithms for synchronizing are presented in section 3 Section 4 closes with a short summary and an overview of future work

2 ARCHITECTURE OF A CLIENT-SERVER-DATA-WAREHOUSE

According to the discussion given above the architecture of the system is equivalent to a client-server-architecture with some specialities due to the usage in a data warehouse environment. The client is the separated computer, which wants to synchronize its data with the changes on the server. In this section three possible architectures are outlined according to [5]. It is assumed that the data on the server are already changed by a separate data-change-process and it is possible to detect the changes without having a historic protocol The details about the necessary data structures for detecting changes and the update process itself are discussed in section 3

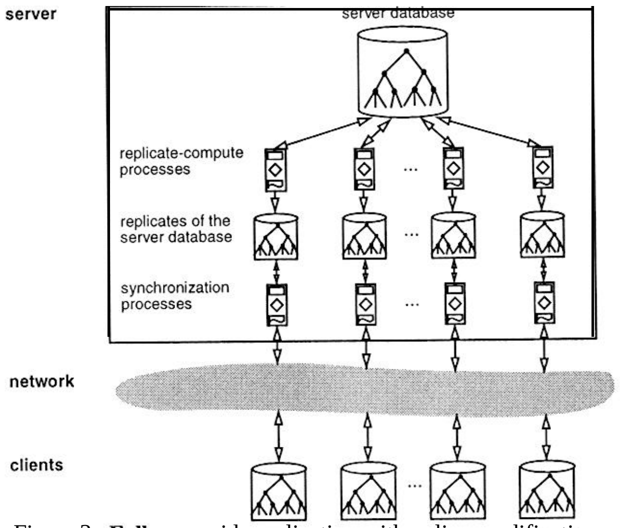
server
server database
replicate-compute processes
replicates of the server database
synchronization processes
network
clients

Figure 3: Full server-side replication with online modification

Final edited form was published in "CIKM01: International Conference on Information and Knowledge Management. Atlanta 2001", S. 69–76, ISBN 978-1-58113-437-7
https://doi.org/10.1145/512236.512246

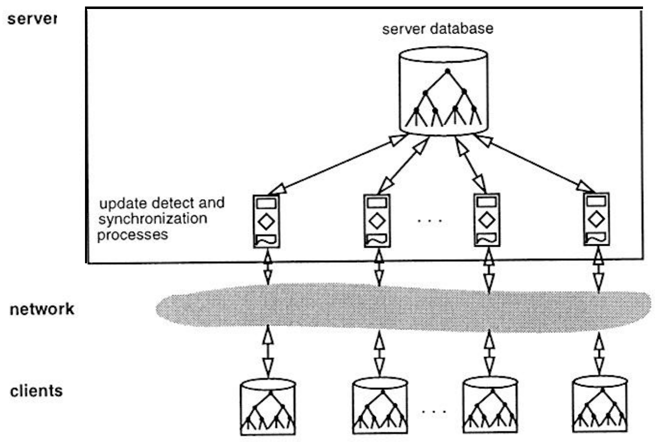

Figure 4: Online computation and synchronization of **modified data**

2.1 Full Server-Side Replication with Online Modification

The first architectural approach shown in figure 3 is build up as follows: After changing the data on the server (server-update-phase. figure I) a process generates replicates of parts of the server database for each client and stores the replicates at the server during the replicate-compute-phase (s figure 1). Each replicate contains only such data, which are interesting for one client or a set of clients with the same data in their local database Therefore, one replicate does not need to be a complete image of the server database. In fact it only contains the data, which have to be stored at the client During the synchronization process (third phase of figure 1), which starts after establishing the connection between client and server, all data at the client are removed at first After that the new data (the corresponding replicate) are transferred from the server to the client and are directly loaded into the client database An imaginable optimization might he that the client downloads the replicates and updates the local database after disconnecting from the sewer The final step is to tag the replicate on the server as downloaded by the client This is a reasonable optimization: Before downloading any data the client checks, whether it has already updated the local database In this case nothing has to he done by the client; otherwise the update process is started

The advantage of this architecture is that the power of the sewer is used to generate the replicates, which can be loaded from the server to the client without any delay Furthermore, the client only needs to load the data into the local database Although the bandwidth and the transfer rate of the network is powerful, the disadvantage of transferring huge data volumes might become a serious problem

2.2 Online Computation and Synchronization of Modified Schema Data

While a sewer process computes replicates of the database in the architecture presented in section 2.1 the architecture discussed in this section provides an online computation and transfer of modified data Figure 4 illustrates the relevant parts and processes of this architectore After establishing the connection between client and server a synchronization process on the server is started to detect the changes, which are relevant to the client This step reflects the replicate-compute-phase shown in figure 1 The detected changes are transferred from the sewer to the client during the synchronization phase (figure I), where the local database is online changed according to the received informations. As soon as all changes are detected and transferred the client disconnects from the server

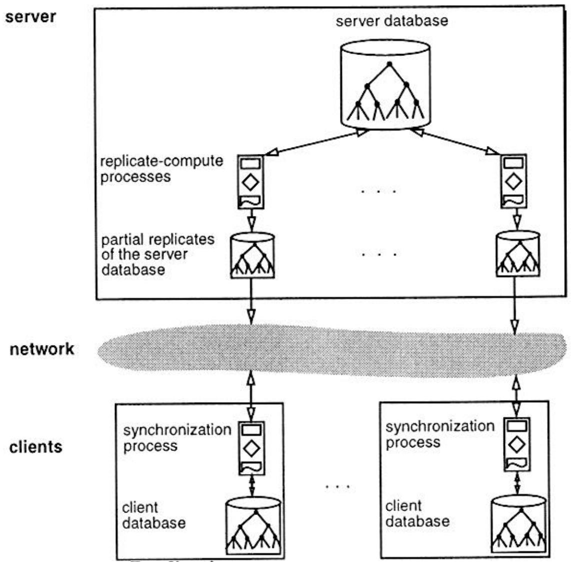

Figure 5: Replication of modified data with offline synchronization

The advantage of this architecture is that instead of a full client replicate only the changed data are transferred from the server to the client Secondly, memory savings are tremendous as no replicate is stored on the server That is why there does not exist any server process that computes the replicates and causes a heavy load on the server at synchronization time

2.3 Replication of Modified Data with Offline Synchronization

This architecture might he considered a combination of the preceding two architectural approaches while combining their advantages (figure 5) According to section 2 1 the idea is that replicates of the server database are computed and stored on the server during the replicate-compute-phase (figure 1); however, in contrast to the approach in section 2.1 only information of the modifications are stored instead of the full replicate After connecting to the server the client downloads the changes Thereafter, the replicate is marked as downloaded by the client, because the client can check on connecting next time, whether it has already updated the local database. Finally, the client disconnects from the sewer and updates the local database. These steps are executed during the synchronization-phase (figure I) and are similar to the proceeding approach of section 2 2

The biggest advantage of this strategy can be seen the reduction of communication cost due to the small amount of transferred data. Furthermore, only small replicates arc stored on the server and the server process can compute these replicates in an offline mode

3 ALGORITHMS FOR UPDATING SCHEMA DATA

When replicating data objects they can become outdated if updates occur. Those updates may concern the data itself as well as the schema of the data. In data warehouse environments data updates are comparatively unproblematic as existing data values remain unchanged and just new data are added Problems arise with derived

Final edited form was published in "CIKM01: International Conference on Information and Knowledge Management. Atlanta 2001", S. 69–76, ISBN 978-1-58113-437-7
https://doi.org/10.1145/512236.512246

data like materialized views. Efficient algorithms of incrementally updating dependent views is still a major research topic although a lot of work has already been done. On the other hand schema changes may happen in classification hierarchies if - for example - new products are introduced or the product classification is restructured if the products are rearranged.

This section presents data structures for updating schema data and explains the update process in detail. An overview of the update process is already given in section 1. The individual steps depend on the selected architecture (section 2). As the approach presented in section 2.3 seems to be the best alternative, all ongoing explanations are referred to this architecture. However, since the differences of each step between the architectures are very small only the update steps for the last architecture are discussed in detail. The transfer to the other architectures is quite simple and, therefore, omitted. In the following the necessary data structures and modification operations, which are the reason for synchronizing, are explained and illustrated by a few examples (section 3.1). The second subsection deals with the most important and complex part of the update process, the detection of changes and the construction of the update structure (replicate-compute-phase, figure 1), which is transferred from the server to the client.

3.1 Data Structures and Modification Operations

This section gives an overview of the data structures necessary to track the schema changes on the server and build update information for the client replicates. The objective of the presented method is to enable server-side schema changes and client updates. When synchronizing with the server the clients should be able to retrieve only the latest state. This means a schema evolution mechanism is implemented. As a full history of all modifications on the server needs not to be available at the client, the data structures holds only the latest state of the changed schema data. Therefore, intermediate states being outdated are not taken into account and are not transferred to the client. Moreover an evolution mechanism is imaginable where historical configurations are retrievable. This would be subject of future work.

Data Structures

To be able to modify classification nodes themselves and their relationship to other classification nodes, a time stamping mechanism for both the nodes as well as the edges has to be introduced. These time stamps are logical time stamps and represent the last modification time ([15]). As new nodes may be introduced or deleted as well as disabled or enabled there is also a valid flag which indicates if the classification node is currently a valid part of a classification hierarchy. On client side the nodes are not disabled and enabled but always deleted and newly created. That means nodes deleted on the server remain there until this information is propagated to all clients. For a description of changed parent-child relations in the classification tree the edges are also times stamped and have a valid flag similar to that of nodes. Figure 6 gives an overview of the data structures for nodes and edges.

```
type node is record {                  type edge is record {
    nodename     string                    parentnode    string
    validTag     boolean                   childnode     string
    lastChange   timestamp                 validTag      boolean
}                                          lastChange    timestamp
                                           changedSubtreetimestamp
                                       }
```

Figure 6: Data structures for nodes and edges

Furthermore each edge has another time stamp 'changedSubtree' which indicates whether modifications in the subtree below the current edge occurred. If an update of a node or an edge in the classification hierarchy was done all edges on the path from the modified object to the root node of the complete tree are marked with the current time stamp. This way all changes in a hierarchy can be tracked by starting at the root and traversing all subtrees with time stamps newer than that of the last synchronization. The server also holds a structure with time stamps of the last synchronization time of each client to be able to extract all changes since the last update.

As the clients usually just have a part of the server schema, the nodes to be transferred to the clients have to be specified. This is done by so called negative-lists identifying all nodes which are not accessible by the client. If a node is an element of a negative-list, the node itself including the whole subtree from this node to the leaf nodes is excluded from synchronization. The server holds a negative-list for each client to regulate the access.

In figure 7(a) a possible server-side classification hierarchy is shown. A negative-list for one client could contain the nodes C and E. The resulting classification tree for the client can be seen in figure 7(b). The nodes together with their subtrees are excluded from the client hierarchy. When testing if a given node N is member of the client hierarchy the tree is run through from N to the root node. At every node **K** it is checked if the node is included in the negative-list. If **K** is member of the negative-list node N cannot be in the resulting tree and the pass is stopped.

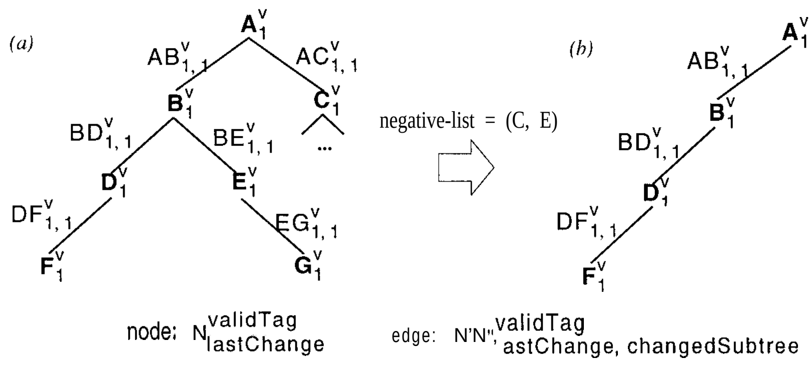

Figure 7: Example of the effect of a negative list

Final edited form was published in "CIKM01: International Conference on Information and Knowledge Management. Atlanta 2001", S. 69–76,
ISBN 978-1-58113-437-7
https://doi.org/10.1145/512236.512246

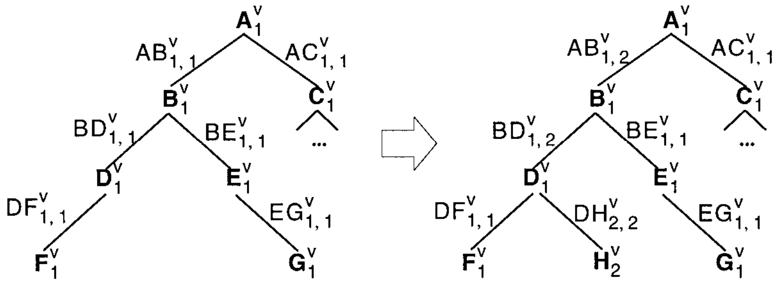

Figure 8: Inserting a new classification node

Figure 7 also shows the notation for nodes and edges used in this paper. The nodes are annotated with a valid tag and a time stamp of the last modification. The edges additionally have a time stamp to indicate the modification in the subtree with the node at the end of the edge as the root of the subtree.

Modification Operations

The remainder of this section describes the possible operations to modify the server side data schema and the resulting changes in the data structures. Modification operations are insertion of a new classification node, deletion of a node and reconnecting a node to a new parent.

When inserting a new classification node the node gets a time stamp with the current logical time and is set valid. Furthermore a new edge is implicitly generated to the specified parent node. The new edge is also marked valid and gets the current time. Ail edges on the path from the inserted node to the root node get a new value for the `changedSubtree` time stamp to efficiently find changed objects in the tree. Figure 8 shows an example of the insertion of a new node. At time 2 the node **H** is inserted into the classification hierarchy. It can also be seen that a new edge **DH** to the parent node **D** is created.

When deleting a node it is marked as inactive and gets the current time stamp. The same holds for the edge connecting this node to its parent. If the deleted node is not a leaf node then the subtree with the deleted node as root is also marked as deleted. Again the path to the deleted node is marked with time stamps. In figure 9 node G is removed which means the node and its connecting edge is set inactive and gets the current time stamp 3.

If a node should be connected to a new parent node the old edge is set inactive and the time stamp is updated as well as a new edge to the new parent node is created which is set active with the current time stamp. The paths to the two modified edges have to be time stamped to be recognizable as modified. This process is illustrated in figure 10 where node **H** is reconnected from parent node D to the new parent **E.** The old edge **DH** gets inactive whereas the new edge **EH** is introduced. The data structures of the node itself remain untouched.

Before closing this section it is annotated that the performance of the algorithms is linear to the height of the tree, because in the worst case starting from a leaf node all parent nodes and edges up to the root node have to be processed and the tags and time stamps have to be updated.

3.2 Processing the Update

This section explains the computation of a replicate for a client whereby the replicate only consists of the changes being relevant to the client. For that an algorithm is presented which can detect the changes inside the classification tree by using the annotations of edges and nodes discussed in the previous section. An alternative could be a historical list of all changes on the server database. However, as we intend to transfer as little data as possible from the server to the client such a list could contain more information than necessary: Firstly, the client is only interested in a part of the classification tree; the root of the subtrees being not interesting are members of the negative-list. Therefore, a full list contains informations of changes of subtrees being of no interest. Secondly, between two synchronizations of the client an edge or node could be changed (i.e. inserted, deleted and modified) several times. But only the last change which reflects the current valid classification tree **must** be transferred to the server. For these reasons an algorithm for detecting changes and the data structure for describing and transferring the changes to the client is explained in the following.

Figure 11 shows the algorithms. At first `computeReplicate` is called with the identification c of the client as parameter. The function initializes and returns a list of changes whereby an XML notation is used (s. below in this section). Obviously if the current time stamp and the last synchronization time stamp are equal, nothing has to be done. Otherwise the function `createReplicate` is called with the root of the classification tree as first parameter, c and the 1 is t as the other parameters.

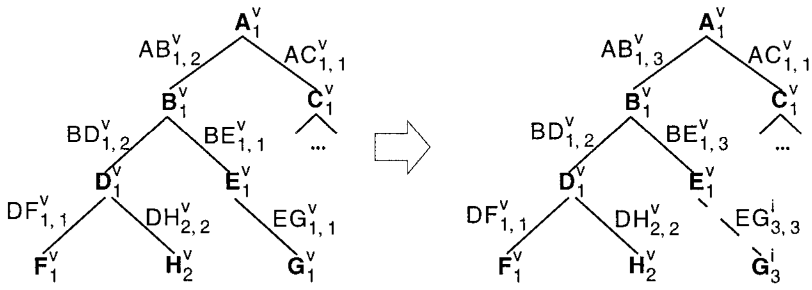

Figure 9: Deleting a classification node

Final edited form was published in "CIKM01: International Conference on Information and Knowledge Management. Atlanta 2001", S. 69–76, ISBN 978-1-58113-437-7
https://doi.org/10.1145/512236.512246

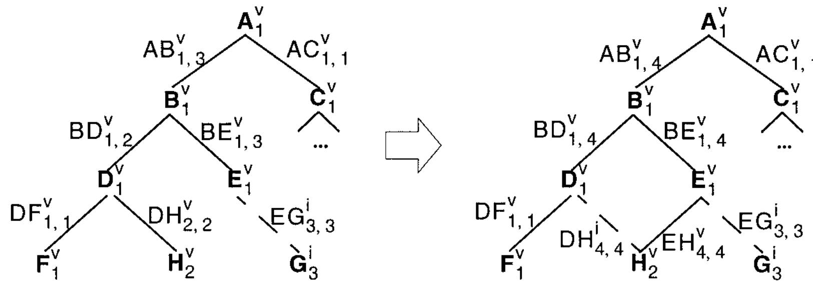

Figure 10: Reconnecting a classification node

The function `createReplicate` has to check the current node at first: If it is a member of the negative-list nothing else has to be done. If the node is changed since the last synchronization point of time the current node is added to the list of changes. The foreach-loop checks for each edge starting from the current node to a child whether the edge is changed. In this case the edge is also added to the list of changes. If the time stamp of the last synchronization of the client is equal to the last change time of the current edge and to the time stamp, which marks the changes in the subtree, this subtree need not to be investigated, because no further changes are done in the subtree. Otherwise the function `createReplicate` is recursively called with the end node of the current edge, the identification of the client and the return list.

For a better illustration of the algorithms we refer to the trees shown in figure 13. Figures (a) and (b) show current valid classification tree at two points of time while figure (c) illustrates the full classification tree on the server which contains all nodes and edges including the time stamps explained in section 3.1: It is assumed that a specific client has synchronized at point of time 1, which reflects the initial state. Up to the next synchronization at point of time 2 several changes are done: At first, the nodes **D, H** and **I** are removed, why the nodes and the edges to the nodes are marked as invalid and the time stamp `lastChange` and `changeSubtree` are set to 2. Secondly, the nodes N and 0 are added to the tree. Finally, the parent of node **F** is reconnected from **B** to C. For the synchronization a replicate of changes is produced for the client. It is assumed, that the negative-list of the client contains the node **E.** Then the replicate-list produced by the algorithm `computeReplicate` (figure 11) contains the following edges and nodes, which are ordered as a result from traversing the tree from the left to the right:

- edges: **DB, DH, DI, BF, CF, CN, NO**
- nodes: **D, H, I, N, 0**

```
Input:      identification of client c
output:     list of edges and nodes in XML

XML computeReplicate ( c ) (
    XML listOfChanges = {};                  //list of changes in XML
    if ( clientSyncTime[c] == curTime )      //client is uptodate
        break;
    else                                     //due to changes compute replicate
        listOfChanges = createReplicate ( root,, c);
    return listOfChanges;
}

Input:      curNode: current node
            c: identification of client
            list: XML-list
output:     list of edges and nodes in XML

XML createReplicate ( curNode, c, list) (
    if ( curNode ∈ negativeList )
        return;
    if ( curNode_lastChange > clientSyncTime[c] )
        addReplicateList ( curNode, list);
    foreach curEdge in successorEdges ( node ) {
        if ( curEdge_lastChange > clientSyncTime[c] )
            addReplicateList ( curEdge, list);
        if ( curEdge_lastChange ==
            curEdge_changedSubtree ==
            clientSyncTime[c] )
            continue;
        createReplicate ( curEdge_childnode, c, list);

    return list:
}
```

Figure 11: Algorithms for computing the replicate

For transferring these information in a standard format an XML representation is used. Figure 12 shows the corresponding XML-DTD; appendix A presents the appropriate XML-Schema. The two main parts of the specification are edge and node each with a few attributes. As the XML-Document for the example above is obvious it is left out.

The last step to be done on the server-side is to update the array holding the last synchronization time of the client. The value is set to the current time stamp.

```
<?xml version="1 .0" encoding="UTF-8"?>
<!ELEMENT ReplicateList (edge I node)'>
<!ELEMENT edge (#PCDATA)>
<!ATTLIST edge
    parentnode  CDATA  #REQUIRED
    childnode   CDATA  #REQUIRED
    validTag (valid I invalid) #REQUIRED
    lastChange CDATA #REQUIRED
    changedSubtree CDATA #REQUIRED
>
<!ELEMENT node (#PCDATA)>
<!ATTLIST node
    node-name   CDATA  #REQUIRED
    validTag (valid I invalid) #REQUIRED
    lastChange CDATA #REQUIRED
>
```

Figure 12: XML-DTD for transferring the replicate

4 SUMMARY AND FUTURE WORK

This paper deals with the efficient synchronization of multidimensional schema data in a client server environment. Inspired by the increasing power of personal computers and the raising bandwidth of networks, transferring parts of the data warehouse to the client, storing the data and computing queries at the client seems to be possible. If additionally the data and structures at the server are changed, then the problem of efficiently detecting and transferring the modifications to the server comes up. The focus and main con-

Final edited form was published in "CIKM01: International Conference on Information and Knowledge Management. Atlanta 2001", S. 69–76, ISBN 978-1-58113-437-7
https://doi.org/10.1145/512236.512246

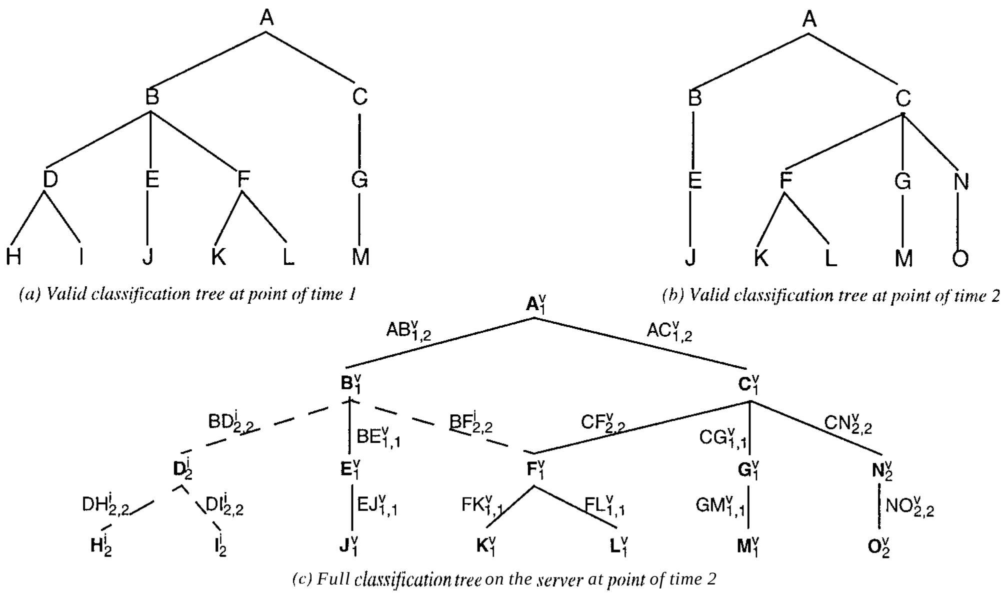

Figure 13: Classification tree at two points of time

tribution of this paper is the presentation of an algorithm for detecting modifications in the classification tree by not having a historical list containing all changes.

After an introduction in section 1 including an overview of the synchronization process several possible architectures are presented (section 2). In section 3 data structures, modification operations and an algorithm for detecting the changes are presented. Although this paper addresses some problems and presents first solutions, many other issues are open, which are part of our future work. At first we think about improving the proposed synchronization technique. The introduced technique provides an image of the current valid and interesting classification tree of the server database. However, we might envison a scenario where the client holds a full replicate of the interesting tree including all changes on the server. This means that all historical informations are stored and transferred to the client. Such a historical list is not necessary for our approach. If it is introduced, then the classical star- or snowflake-schema ([13]) have to be extended. For instance from a relational point of view nested tables ([16]) could be used to store the historical informations for each node. Another point of our future work is the following: It is imaginable, that the classification structure of the client is changed and propagated to the server. If the server structure is not changed then the proposed algorithm can be used. Otherwise transaction management with locks of nodes and subtrees has to be introduced according to [7]. Finally in future the performance of the proposed algorithms should be evaluated in practice.

Summarizing the paper we think that the concept of having parts of the data warehouse stored on a personal computer becomes an significant fact in future, because the subject 'information' becomes more and more important even for people being not permanent connected to the central data warehouse.

REFERENCES

1 Agrawal, R.; Gupta, A.; Sarawagi, S.: Modeling Multidimensional Databases, in: *Proceedings of the 13th International Conference on Data Engineering* (ICDE'97, Birmingham, Großbritannien, April 7-11), 1997, pp. 232-243

2 Baralis, E.; Paraboschi, S.; Teniente, E.: Materialized View Selection in a Multidimensional Database, in: *Proceedings of the 23rd International Conference on Very Large Data Bases* (VLDB'97, Athen, Griechenland, August 25-29), 1997, pp. 156-165

3 Castro de, C.; Grandi, F.; Scalas, M.: Schema Versioning For Multitemporal Relational Databases. In: *Information Systems* 22(5), 1997, pp. 249-290

4 Chaudhuri, S.; Dayal, U.: An Overview of Data Warehousing and OLAP Technology, in: *ACM SIGMOD Record 26(1997)1*, pp. 65-74

5 Cohen, N.H.: *The MVCRS Java Framework for Mobile Data Synchronization*, IBM Research Report RC 21774 (98049), August 16, 2000

6 Gadia, S.; Nair, S.: Temporal databases: a prelude to parametric data. In: *Temporal Databases: Theory, Design, and Implementation.* Benjamin/Cummings, 1993, pp. 28-66

7 Gray, J.; Reuter, A.: *Transaction Processing: Concepts and Techniques.* San Mateo (CA): Morgan Kaufman Publishers, 1993

8 Gyssens, M.; Lakshmanan, L.V.S.: A Foundation for Multi-Dimensional Databases in: *Proceedings of the 23th International Conference on Very Large Data Bases* (VLDB'97, Athen, Griechenland, August 25-29), 1997, pp. 106-115

9 Harinarayan, V.; Rajaraman, A.; Ullman, J.D.: Implementing Data Cubes Efficiently, in: *Proceedings of the International Conference on Management of Data* (SIGMOD'96, Montreal, Quebec, Canada, June 4-6), 1996, pp. 205-216

10 Hurtado, C. A.; Mendelzon, A. 0.; Vaisman, A. A.: Maintaining Data Cubes under Dimension Updates. In: *Proceedings of the 15th International Conference on Data Engineering* (ICDE'99, Sydney, Australien, March 23-26), 1999, pp. 346-355

Final edited form was published in "CIKM01: International Conference on Information and Knowledge Management. Atlanta 2001", S. 69–76,
ISBN 978-1-58113-437-7
https://doi.org/10.1145/512236.512246

11 Hurtado, C. A.; Mendelzon, A. 0.; Vaisman, A. A.: Updating OLAP Dimensions. In: *Proceedings of the 2nd International Workshop on Data Warehousing and OLAP* (DOLAP'99, Kansas City, Missouri, USA, November 6), 1999, pp. 60-66

12 Inmon, W.H.: *Building the Data Warehouse.* John Wiley & Sons, New York et.al., 1996

13 Kimball, R.: *The Data Warehouse Toolkit.* John Wiley & Sons, New York et.al., 1996

14 Kimball, R.: Slowly changing dimensions. Unlike OLTP systems, data warehouses can track historical data. In: *DBMS Online 9* (4), *1996*

15 Lamport, L.: Time, Clocks and the Ordering of Events in a Distributed System. In: *Communications of the ACM, 21(7),* 1978, pp. 558-565

16 Oracle, http://www.oracle.com/, 2001

17 Roddick, J. F.; Craske, N. G.; Richards, T. J.: A Taxonomy for Schema Versioning Based on the Relational and Entity Relationship Models. In: *Proceedings of the 2th International Conference on the Entity-Relationship Approach* (ER'93, Arlington, Texas, USA, Dezember 15-17), 1993, pp. 137-148

18 Roddick, J. F.: A Model for Schema Versioning in Temporal Database Systems. In: *Proceedings of the 19th ACSC Conference* (ACSC'96, Melbourne, Australien, January 3 1 - February 2), 1996, pp. 446-452

19 Sapia, C.; Blaschka, M.; Höfling, G.: *An Overview of Multidimensional Data Models for OLAP.* FORWISS Report FR-1999-001, 1999 (electronic version: http://www.forwiss.de/public/ reports.html)

APPENDIX A: XML-SCHEMA FOR TRANSFERRING THE REPLICATE

```
<?xml version="1.0" encoding="UTF-8"?>
<!--W3C Schema generated by XML Spy v3.5 NT (http://www.xmlspy.com)-->
<xsd:schema xmlns:xsd="http://www.w3.org/2000/1 0/XMLSchema" elementFormDefault="qualified">
    <xsd:element name="ReplicateList">
        <xsd:complexType>
            <xsd:choice minOccurs="0" maxOccurs="unbounded">
                <xsd:element ref="edge"/>
                <xsd:element ref="node"/>
            </xsd:choice>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="edge">
        <xsd:complexType>
            <xsd:simpleContent>
                <xsd:restriction base="xsd:string">
                    <xsd:attribute name="parentnode" type="xsd:string" use="required"/>
                    <xsd:attribute name="childnode" type="xsd:string" use="required"/>
                    <xsd:attribute name="validTag" use="required">
                        <xsd:simpleType>
                                <xsd:restriction base="xsd:NMTOKEN">
                                        <xsd:enumeration value="valid"/>
                                        <xsd:enumeration value="invalid"/>
                                </xsd:restriction>
                        </xsd:simpleType>
                    </xsd:attribute>
                    <xsd:attribute name="lastChange" type="xsd:string" use="required"/>
                    <xsd:attribute name="changeSubtree" type="xsd:string" us&required"/>
                </xsd:restriction>
            </xsd:simpleContent>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="node">
        <xsd:complexType>
            <xsd:simpleContent>
                <xsd:restriction base="xsd:string">
                    <xsd:attribute name="node-name" type="xsd:string" use="required"/>
                    <xsd:attribute name="validTag" use="required">
                        <xsd:simpleType>
                                <xsd:restriction base="xsd:NMTOKEN">
                                        <xsd:enumeration value="valid"/>
                                        <xsd:enumeration value="invalid"/>
                                </xsd:restriction>
                        </xsd:simpleType>
                    </xsd:attribute>
                    <xsd:attribute name="lastChange" type="xsd:string" use="required"/>
                </xsd:restriction>
            </xsd:simpleContent>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```