Proc. BTW 2003, Leipzig, Februar 2003

Efficient Assembly of Product Structures in Worldwide Distributed Client/Server Environments

E. Müller, P. Dadam
University of Ulm
Faculty of Computer Science
Dept. Databases and Information Systems
{mueller,dadam@informatik.uni-ulm.de}

M. Feltes
DaimlerChrysler
Research and Technology
Dept. RIC/ED, Ulm
{michael.feltes@daimlerchrysler.com}

Abstract: The efficient management of product-related data is a big challenge for many manufacturing companies, especially the larger ones like DaimlerChrysler. So-called Product Data Management (PDM) systems are used to tackle this task. But especially in worldwide distributed product development environments, where low bandwidth networks are used, response times of PDM systems often raise to several minutes even for simple user actions. The main reason for this is the amount of wide area network communication caused by poor implementations of PDM systems which a) often use the underlying database management system like a stupid record manager, and b) do hardly know anything about the distribution of the data. In this paper we introduce an extension of directed acyclic graphs which enables us to adequately represent product structures. On the basis of this representation we show how some additional information describing the data distribution can be used to assemble distributed product structures very efficiently.

1 Introduction

Product development is a time-consuming and costly process. Keen competition especially forces the manufacturing companies to shorten this process in order to survive. Besides the wellknown CAD and CAE tools, which help to optimize the completion of *intra*-disciplinary tasks, so called Product Data Management (PDM) systems have been established during the last years in order to support the *inter*-disciplinary tasks at best (cf. [MOn], [MP], [WT], [CIM97]).

PDM systems claim to be the information backbone serving all users involved in the development process (cf. [OLR95]). This means that all data relevant to somebody in this process have to be managed by the PDM system – beginning at the core product structure which describes the composition hierarchy of parts and subparts, and ending at all describing documents like specifications, CAD files, work orders, simulation results, and so on (cf. figure 1). Especially in case of complex products like cars, PDM systems have to manage thousands of such objects and provide these data to thousands of users – with acceptable response times!

It is not only the amount of data which makes a product complex. It is also the fact that products can be customized or *configured* and therefore typically exist in several different

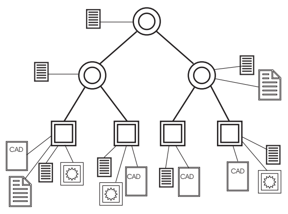

Figure 1: A simplified product structure with several document types

versions: A car for example may be offered in a "basic model" which can be configured according to the customers' demands by equipping it with lots of additional features, e. g. a sunroof, a navigation system, xenon headlights and so on. Easy to imagine that the number of different producible cars can be extremely high, but the difference between two configurations may be rather small as all configurations share a lot of common parts. So instead of storing each configuration separately (which would lead to totally inefficient structure handling) the structures of all supposable product instances are combined in one large structure. To retrieve the structure of a certain product instance, parts of this structure may be ruled out by appropriate conditions.

A typical way of using PDM systems is to *navigate* in the product structure according to some configuration options (like the features in the car example). Users begin at the top-level element of the product structure (i. e. the product itself) and expand the subjacent level of the structure. This so-called *single-level expand* is repeated until the user finds what he looks for. In doing so, today's PDM systems – for various reasons they typically sit on top of relational database management systems – use SQL as a simple record manager: The navigational traversal of the product tree is typically translated nearly one-to-one into single isolated SQL queries. This stepwise navigation also works for the *multi-level expand* which expands the entire product structure by recursively applying the single-level expand method.

A closer look to the representation of product structures within the database shows how these expand actions work: Typically parts consist of several subparts (which are parts as well), and sometimes one subpart may be integrated into several parts. This recursive n:m-relationship is realized by so-called link-objects. In distributed environments the link-objects also identify the databases where the related parts are stored. In today's PDM systems an expand action for a certain part first retrieves all link-objects that identify the corresponding subparts. In a second step the subparts are retrieved from the database(s) indicated in the link-objects.

This *naive (one-object-at-a-time* or *one-level-at-a-time)* approach leads to a large number of SQL queries (and result messages, of course). In environments, where the DBMS and PDM system are connected via high-speed netwoks with low latency times, this may not cause too much harm. The picture may change dramatically, however, if the users are working in geographically distributed environments. Some experiments in prototypical but realistic PDM environments at DaimlerChrysler have shown that response times may rise by orders of magnitude, e. g. from 1–2 minutes in the local context to 30 minutes in the "intercontinental" context. The reasons for these inacceptable response times are the typically long latency times and low bandwidth of wide area networks.

Obviously, the aim must be to cut down response times of expansion operations by a solution that a) minimizes the number of communications (i. e. the number of SQL queries) between distributed locations, and b) in doing so, does not increase the volume of transmitted data. A very promising way to achieve this goal is to provide a function shipping approach instead of today's stepwise data shipping.

In [MDEF01] we describe a first approach for pushing the navigational process from the PDM system to the DBMS. Substituting several isolated consecutive SQL queries by one recursive SQL query (cf. [ANS99], [EM99], [IBM01]) may dramatically speed up the expand methods in environments having only one central data server and several distributed PDM servers. However, if the data is stored distributedly – i. e. the tables storing parts and link-objects are split into several partitions which are distributed across several database servers – there is no way to efficiently assemble the entire product structure by one single recursive query: At query generation time there is no information available about the partitions that have to be accessed for assembling the given part. This information can only be obtained by interpreting link-objects. As SQL does not provide the ability to dynamically switch to different data sources during query execution according to such "run-time interpreted" data, the recursive query either considers a) only one partition or b) the union of all partitions. In the first case one needs to generate a sequence of recursive queries each of which only collects the parts and link-objects stored in one partition. In the latter case too many tuples, which do not have any effect on the resulting structure, are transmitted to the executing server.

In principle, the first approach is the basis of the solution presented in this paper. We provide additional information about which parts at which servers have to be retrieved when assembling the product structure. This index-like information, the so-called *object link and location catalog*, enables us to contact each server involved in a product structure assembly only once. It avoids multiple retrieval of multiply occurring parts within the structure, it allows queries to different database servers to be executed in parallel, and it helps to minimize the number of communications between database servers.

The rest of this paper is organized as follows: Section 2 introduces an extension of a common directed acyclic graph, the so-called *directed acyclic condition graph*. This enables us to adequately represent configurable product structures. In section 3 we define the object link and location catalog. Usage and benefits of this catalog are described in section 4. The algorithms for handling the catalog are presented in section 5. Related work is discussed in section 6. Section 7 concludes and gives an outlook to further work.

2 Product Structure in Terms of a DAG

2.1 Representing Product Structures

At first glance a product structure can be adequately represented by a simple directed acyclic graph (DAG, cf. [CLR96],[Sed88]): Each project, part and subpart is a vertex of such a graph, and any directed relation between them – e.g. the "part-uses-subpart" relation – is a corresponding edge. But unfortunately, the simple DAG lacks the ability to express the flexibility of a product structure regarding product configuration and structure options (confirm section 1). It would be neccessary either to store each product instance in an isolated DAG (which is impossible for products with a large number of possible configurations like cars) or to simply "merge" the structures of all producible product instances losing the capability to correctly extract the structure of a certain product instance. In order to solve this problem we need the ability to express whether a certain edge of the DAG is a member of an actual corresponding product instance or not. This can be achieved by extending the DAG by appropriately defined configuration conditions.

2.2 Extending DAGs for Product Structure Requirements

We assume $\mathcal{O}$ to be a set of available options $\mathcal{O} = \{\chi_1, \chi_2, \ldots, \chi_n\}$ that can be used to configure product instances. A user can choose some, all or none of these options in order to indicate which optional parts have to be included into the basic product structure. We describe this situation by the following formalism in the style of propositional calculus (cf. [Fit90], [LE78]):

$\mathcal{O}$ can be viewed as a set of atomic formulas. When the user chooses some options, he defines an *assignment* $\bar{\mathcal{A}} : \mathcal{O} \to \{0, 1\}$, i. e. selected options are assigned the value 1, the others are set to 0.

Dependent on $\mathcal{O}$ we introduce the set $\mathcal{C}$ of logic expressions – called *conditions* – which are solely built over the atomic formulas in $\mathcal{O}$, i. e. the elements of $\mathcal{O}$ can be combined with the operators AND ($\wedge$), OR ($\vee$) and NOT ($\neg$). For completeness reasons we assume the "true condition" $\top$ (always true) to be an element of $\mathcal{C}$, too. Now we extend $\bar{\mathcal{A}}$ to $\mathcal{A} : \mathcal{C} \to \{0, 1\}$ which indicates whether an expression $c \in \mathcal{C}$ evaluates to true (i. e. 1) or not.

Definition 1: Directed Acyclic Condition Graph (DACG)

> A *D̲irected A̲cyclic C̲ondition G̲raph* G^c is an extended DAG defined as follows:
>
> $G^c = (V, E, \mathcal{C})$ where $E \subseteq V \times V \times \mathcal{C}$, V is the set of vertices, $\mathcal{C}$ is a set of conditions, and E is the set of condition-annotated edges between vertices.

The semantics is as follows: An edge from vertex u to vertex v is only valid, if the annotated condition c evaluates to TRUE (i. e. $\mathcal{A}(c) = 1$) for a given assignment $\bar{\mathcal{A}}$.[1] In general we denote this edge $u \overset{c}{\rightharpoondown} v$, but if $c = \top$ we omit the condition and write simply $u \rightharpoondown v$. Similarly, a path from vertex u to vertex v in a DACG G^c is written as $\langle u \overset{c_1}{\rightharpoondown} n_1 \overset{c_2}{\rightharpoondown} \ldots \overset{c_k}{\rightharpoondown} n_k \overset{c_{k+1}}{\rightharpoondown} v\rangle$.

For our investigations in subsection 3.2 we introduce the transitive closure of a DACG here:

Definition 2: Transitive closure of a DACG

> The transitive closure G^{c*} of a DACG $G^c = (V, E, \mathcal{C})$ is defined as follows:
>
> $G^{c*} = (V, E^*, \mathcal{C}^*)$ where $E^* = E \cup \{(u \overset{c}{\rightharpoondown} v) | u, v \in V \wedge$
> $\exists d = \langle u \overset{c_1}{\rightharpoondown} n_1 \overset{c_2}{\rightharpoondown} \ldots \overset{c_k}{\rightharpoondown} n_k \overset{c_{k+1}}{\rightharpoondown} v\rangle$ in $G^c, (k > 0)$,
> and $c = c_1 \wedge c_2 \wedge \ldots \wedge c_{k+1}\}$ and $\mathcal{C}^* = \mathcal{C} \cup \{c | u \overset{c}{\rightharpoondown} v \in E^*\}$.

Like in ordinary DAGs an edge of a transitive closure of a DACG connects vertices that are starting and ending points of a path in the original graph. In case of DACGs each edge of the transitive closure is annotated with the conditions which are related to the edges of the corresponding path, connected by the "AND" operator.

2.3 Partitioning DACGs

In typical scenarios product structures are distributed accross several partners and suppliers. In such environments it is realistic to assume that each involved partner stores only those portions of the structure which he is responsible for or which correspond to parts he produces. So, looking at the entire product structure as a DACG, each server manages some sort of *partition* of this DACG.

We define a partition P_{s_i} of a DACG as follows:

Definition 3: Partition of a DACG

> Assume $G^c = (V, E, \mathcal{C})$ a DACG and $f : V \to S$ a function that "marks" all the vertices of G^c with an element of S. For each element $s \in S$ there exists one subgraph P_s of G^c where
>
> $$P_s = (V_s, E_s, \mathcal{C}) | V_s = \{v \in V | f(v) = s\} \wedge$$
> $$E_s = \{(u \overset{c}{\rightharpoondown} v) \in E | u \in V_s \vee v \in V_s\}$$
>
> The *connected components*[2] of each P_s are called *partitions* $P_{s_i} = (V_{s_i}, E_{s_i}, \mathcal{C})$ of G^c.

[1]In the automotive industry the condition c for example may guarantee that "v is only part of u if the car in scope has to be equipped with a sunroof".

[2]The *connected components* of a directed graph $G = (V, E)$ correspond to the connnected components of the undirected version $G' = (V, E')$ of G. In the same sense we use the term for a DACG, too.

Remark: a partition P_{s_i} does not represent a DACG as defined in definition 1, since E_{s_i} may contain at least one edge $(u \xrightarrow{c} v)$ where either $u \notin V_{s_i}$ or $v \notin V_{s_i}$! These edges combine two differently marked partitions and belong to both connected partitions!

In our environment S would be the set of servers or PDM systems participating in a product development environment. The function f would map each elementary object (e. g. an assembly) to the server it is stored at.

Figure 2 shows an example of a very simple DACG split into several partitions. In this example, the set of marks is S=$\{A, B, C, D\}$, and the mapping function f assigns the following values: $f(1)$=$f(2)$=$f(3)$=$f(13)$=A, $f(4)$=$f(12)$=B, $f(5)$=$f(6)$=$f(8)$=$f(9)$=$f(10)$=C, $f(7)$= $f(11)$=D. The set of conditions is $\mathcal{C}$=$\{c_1, c_2, c_3, c_4, \top\}$. The partitions of G^c are marked with grey shapes.

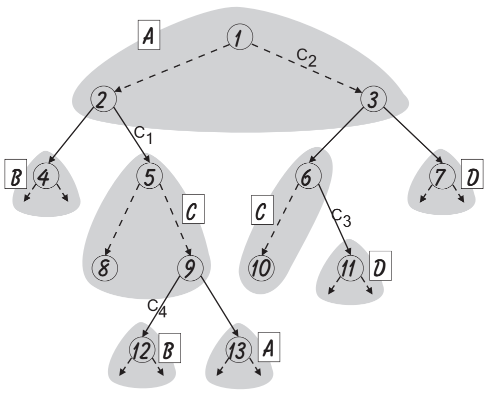

Figure 2: A simple distributed DACG G^c

3 Object Link and Location Catalogs

3.1 Conventional Assembly Strategies

Partitioning as introduced in subsection 2.3 enables us to specify the distribution of a DACG across several servers. In order to reconstruct a suchlike distributed DACG – or, speaking in terms of product structures, to assemble an entire product of all parts and subparts – there exist several well-known, conventional strategies (cf. [MGS$^+$94]):

1. Starting with the top-level element (the root node of the graph), fetch recursively all directly related subparts of the parts already retrieved (*one-level-at-a-time*). – As already outlined, this is the naive, very inefficient approach.
2. Ship all remote data to the location where the top-level element is stored (or where the action is started) and assemble the entire complex object subsequently at that

site. – This strategy may be very expensive: The resulting complex object may consist of only a very small fraction of all transmitted vertices (i. e. simple objects) due to some user-selected conditions. Unfortunately, the objects not included in the result possibly caused a non-negligible transmission overhead.

3. The server starting the action becomes the "master" of the activity (storing a partition we call P_{master}), all other servers behave like "slaves" (storing $P_{s_1}, \ldots, P_{s_n}$). The master assembles the desired structure locally (as far as possible) and requests missing parts of the structure from the slaves all times it reaches an edge that combines P_{master} with one of the other P_{s_i}. As long as the result of a remote server s_i contains an edge that connects P_{s_i} to P_{s_j}, the master has to request the missing part of the structure from the server storing P_{s_j}.

   To illustrate this approach, we assume S (cf. figure 2) to be the set of participating product data servers. Server A at first retrieves vertices $1, 2$, and 3, interprets the edges $2 \rightarrow 4$, $2 \rightarrow 5$, $3 \rightarrow 6$, and $3 \rightarrow 7$ (to simplify matters we omit conditions here) successively and sends corresponding requests to servers B ("assemble substructure with root 4"), C ("assemble substructure with root 5", "assemble substructure with root 6"), and D ("assemble substructure with root 7). The result of server B does not contain any edges that combine two partitions, but the results of server C do: They contain $9 \rightarrow 12$ and $9 \rightarrow 13$, and $6 \rightarrow 11$ respectively, so the master (i. e. server A) has to request the substructures with roots 12, 13 and 11 from the corresponding servers B, A(!), and D. After this the structure is complete.

   There is one major disadvantage with this approach: Each server may be contacted more than once within *one* activity as it happend to the servers B, C, and D in our example. This causes superfluous net traffic and therefore – especially in wide area networks – leads to long response times.

4. Instead of having only *one* master which requests substructures from remote locations, each server is allowed to do so. In our example server A requests the substructures with roots 5 and 6 from server C which in turn requests the substructure with root 12 from server B, 11 from server D etc. and returns the combined results to A. A finishes the action after receiving the responses of all servers according to their requests.

   This approach has the disadvantage mentioned in approach 3, too. In addition to this, if there exist cyclic dependencies within the servers storing partitions of the DACG (e. g. server A references server C which in turn references A) parts of the graph may be sent through the network multiple times (A sends to C, which returns the combined result back to A). This is absolutely dissatisfying!

5. One could also use a mixture of approaches 3 and 4. A server, for example, which is reachable only through a very slow network connection may not be allowed to request subtrees from remote servers, except it is the master according to approach 3. – This may eliminate some low speed network communications, but the disadvantages cannot be obviated really.

As a matter of fact, using conventional algorithms there is no way to reconstruct a widely distributed DACG causing only minimal communication effort. Obviously, the reason for poor performance is that – depending on the distribution of the DACG – remote servers may have to be contacted multiple times within one single assembly operation. So, what we need is some additional information about the distribution of the DACG to avoid such multiple server connections when accessing more than one partition of the structure. In our example server A initially does not know anything about the edge between vertices 9 and 12, so A does not know that there exists an additional (transitive) dependency from A to B. However, if A knew this, it could request vertex 12 from B directly, saving one communication to B if the requests for vertices 4 and 12 were combined! So what we need is some sort of a "virtual edge" e. g. between vertices 2 and 12. Suchlike edges will be stored in the object link and location (OLL) catalog.

3.2 Definition of OLL Catalog

The challenge is to construct the OLL catalog in that way that it is a) minimal in size (that means it does not contain irrelevant or obsolete information) and b) complete (no information neccessary is missing). Based on the DACG introduced in subsection 2.2 we define the OLL catalog as follows:

Definition 4: OLL Catalog

Assume $G^c = (V, E, \mathcal{C})$ to be a DACG, and $f : V \rightarrow S$ a function for marking vertices. Assume $G^{c*} = (V, E^*, \mathcal{C}^*)$ to be the transitive closure of G^c. Then the *OLL catalog* E' of G^c is defined as:

$$E' = \{u \xrightarrow{c} v \in E^* \setminus E | \exists d = \langle u \xrightarrow{c_1} n_1 \xrightarrow{c_2} \ldots \xrightarrow{c_k} n_k \xrightarrow{c_{k+1}} v\rangle \text{ in } G^c, \\ (k > 0), \text{where} f(n_i) \neq f(u) \wedge f(n_i) \neq f(v), 1 \leq i \leq k\}.$$

In other words: The OLL catalog contains a virtual connection of two vertices u and v if v is reachable (in the original DAG) from u by only traversing vertices that are marked differently from u and v. Figure 3 shows the DACG of figure 2 extended by edges of its corresponding OLL catalog ($2 \xrightarrow{c_1 \wedge c_4} 12$, $2 \xrightarrow{c_1} 13$, and $3 \xrightarrow{c_3} 11$).

4 Using OLL Catalogs

4.1 Description of the Approach

The goal of the OLL catalog is to enable performing of recursive expansions a) in one single path and b) to visit each server involved only once. This is achieved by providing enough information in the OLL catalog such that all "transitive" remote requests can be performed by the initiating server. If, for example, the product structure in figure 3 has to

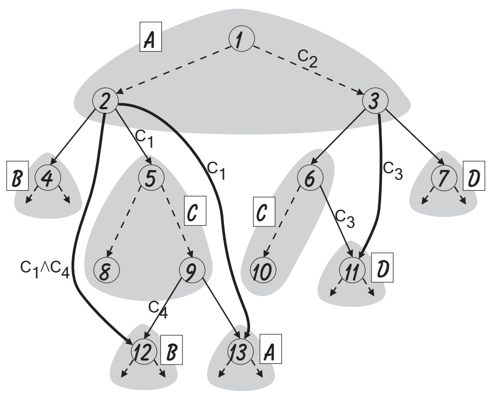

Figure 3: The DACG G^c of figure 2 extended by the edges of the OLL catalog

be expanded by server A, server A will contact servers C, B, and D in order to retrieve the vertices stored there, while servers B, C, and D only perform local expansions.

The multi-level expand of a product structure can be split into three steps: First, upon request the server storing the top-level element of the structure expands the structure according to the locally available structure information. During this recursive descent the server collects unresolved references to remote parts. In the second step, recursive queries are generated for retrieving the remote parts. Third, the queries are executed at the remote sites in parallel and return partially assembled substructures. We will show this procedure by performing a multi-level expand of vertex 1 in the example in figure 3.

We assume c_1=c_2=c_3=c_4=1, i. e. all partitions of the DACG have to be retrieved.

At first, server A expands the structure of vertex 1 locally: The edges pointing from vertex 1 to vertices 2 and 3 are retrieved and interpreted. As vertices 2 and 3 are stored locally, they can be fetched at once. As there are no edges in the OLL catalog starting at vertex 1, server A immediately tries to expand the structure of vertex 2. The corresponding edges are fetched from the local partition of server A, but they point to the vertices 4 and 5 at servers B and C respectively. Instead of retrieving the two remote vertices immediately, server A collects them in a set of unresolved references for later usage.

At this point server A accesses the OLL catalog again: There exist two entries in the catalog, pointing from vertex 2 to vertices 12 and 13 at servers B and A respectively. The conditions attachted to both vertices evaluate to true ($c_1 \wedge c_4$=1, and c_1=1, too), so both vertices are in the scope of the result. OLL catalog entries pointing to a partition that is stored locally are processed immediately, the target vertices of the remaining entries are added to the set of unresolved references. This causes all locally stored partitions, which are reachable from the vertex in scope (in the example this is vertex 1), to be expanded in one recursive descent. So, server A now proceeds by expanding vertex 13 according to the method just described for vertex 2.

After finishing the assembly of vertex 13, server A turns towards vertex 3. All edges in the

DACG as well as in the OLL catalog, which start at vertex 3, point to vertices at remote partitions. Therefore they are collected in the set of unresolved references, too.

At this moment, server A finishes its local recursion. The set of unresolved references contains the following vertices: 4, 12, 5, 6, 11, and 7.

Now the remote servers can be instructed to locally expand the missing parts of the structure: Server B has to assemble the subgraphs of vertices 4 and 12, server C the subgraphs of vertices 5 and 6, and server D the subgraphs of vertices 11 and 7. Note that each server needs to be contacted only once in order to initiate the local expands!

As each of the three remote servers B, C, and D performs the same steps we show the local expansion at server C only. Similar to the procedure already described for server A, the server C tries to expand vertex 5 (vertex 6 either is processed in parallel or after finishing the expansion of vertex 5) using the edges in the local DACG which references vertices 8 and 9. Vertex 8 is a leaf in the graph and therefore cannot be further expanded. In contrast, vertex 9 has references to the vertices 12 and 13 at servers B and A respectively. As vertex 13 is already expanded by server A, and vertex 12 is processed by server B in parallel, the recursive expansion stops here. Finally, the result is returned to server A.

After receiving all subgraphs from the involved remote servers, the entire structure is available at server A.

Further details can be found in the algorithms in section 5.

4.2 Benefit of Using OLL Catalogs

Using the naive *one-object-at-a-time* approach, the expansion of a structure needs as many remote accesses as remote nodes are involved in the structure. In figure 3, if the partitions stored on servers B and D only contain the vertices 4, 12, 7, 11 respectively, then nine remote communications are necessary for the expansion.

The approach using recursive queries without OLL catalogs will require as many remote accesses as remote partitions are involved in the expansion of the structure. In our example this still would lead to six remote communications.

By the usage of OLL catalogs the number of remote communications can be reduced to the number of remote servers participating in the expansion. As each server in our example stores two partitions of the structure, half of the six remote accesses can be saved. Furthermore, in contrast to the other approaches, these remaining three requests can be processed in parallel. Thus the overall expand does not take much longer than the most time-consuming remote expand of a single substructure! If we assume the three remote servers B, C, and D of our example to require similar execution times, the response time can be cut down by approximately another two thirds!

In general, the usage of OLL catalogs reduces the amount of remote accesses to the lowest possible extent, namely the number of database servers storing product data. Obviously the benefit of this approach depends on the distribution of data. If there are only very few servers storing more than one partition of the structure, it may not lead to much less communications. In heavily distributed environments, however, OLL catalogs may save half of the remaining remote communications and even more.

Nevertheless, if the number of communications can not be cut down by the OLL catalog due to a "smooth" data distribution, response times may still dramatically decrease: The recursive queries for assembling substructures on different servers can be executed in parallel since the transitive dependencies between different partitions are resolved by the OLL catalog and already considered when initiating the remote queries.

So, for all cases of distribution, OLL catalogs allow efficient assembly of product structures. Improvements regarding the response times are achieved by reducing the number of communications and by parallel execution of remote queries.

5 Algorithms for Handling OLL Catalogs

The algorithms in this section work on a DACG $G^c = (V, E, \mathcal{C})$. We use some notations we informally introduce here:

S is the set of servers (hosts) which store one or more partitions of G^c.

$f(x)$ is a function $f : V \rightarrow S$.

$z = \langle a, b, c \rangle$ denotes a triple z consisting of the elements a, b, and c. The elements can be accessed via $z.a$, $z.b$ and so on.

$g^{(f(v))}(x)$ means that the function $g(x)$ has to be evaluated at the server storing vertex v.

$s[z.x \leftarrow y]$ denotes an update action on all tuples z within the set s. The value of attribute x is set to y.

OLL denotes the OLL catalog at the current (working) server (i. e. the server that runs the procedure accessing the OLL catalog).

$\mathcal{A}$ describes the structure options in scope (as introduced in subsection 2.2).

5.1 OLL Catalog Initialization

The initialization process of the OLL catalog is distributed across all servers which store parts of the structure. In order to create catalog entries we walk through the entire structure depth-first, starting at the server that stores the root of the structure. At each vertex we test if a OLL edge has to be created to one of the already visited vertices in the path down from the root vertex (cf. definition 4).

The initialization procedure is initially called with `Init(root,`ε`,`$\emptyset$`)`.

Algorithm 1: (Initialization of the OLL catalog)

```
Init(in vertex u, in condition precond, in borders B)
1   new_edges ← ∅
```

```
sorted_B $\leftarrow \emptyset$
if $(B == \emptyset)$ then seq $\leftarrow 1$
else
    sorted_B $\leftarrow$ sort(B) on $B[z.seq\_number]$ desc
    seq $\leftarrow max(B[z.seq\_number]) + 1$
endif
if $(precond == \varepsilon)$ then
    for all $\langle v, c, q\rangle \in sorted\_B$ do
        if $(v \xrightarrow{c} u \notin E)$ then
            $OLL \leftarrow OLL \cup \{v \xrightarrow{c} u\}$
            if $(f(u) == f(v))$ then
                break
            else
                new_edges $\leftarrow$ new_edges $\cup \{v \xrightarrow{c} u\}$
            endif
        endif
    enddo
endif
for all $u \xrightarrow{c} p \in E$ do
    succ.push $\langle u \xrightarrow{c} p, precond\rangle$
enddo
while $(succ.test\ != \mathrm{NULL})$ do
    $\langle u \xrightarrow{c} v, cond\rangle \leftarrow succ.pop$
    if $(f(u) == f(v))$ then
        for all $v \xrightarrow{\bar{c}} w \in E$ do
            succ.push $\langle v \xrightarrow{\bar{c}} w, cond \wedge c\rangle$
        enddo
    else
        $B' \leftarrow (B[z.cond \leftarrow z.cond \wedge cond \wedge c] \backslash$
            $\{b \in B : f(b) == f(u)\}) \cup \{\langle u, c, seq\rangle\}$
        succ_edges $\leftarrow$ Init$^{(f(v))}(v, \varepsilon, B')$
        for all $x \xrightarrow{c} y \in$ succ_edges do
            if $(x == u)$ then
                $OLL \leftarrow OLL \cup \{x \xrightarrow{c} y\}$
            else
                new_edges $\leftarrow$ new_edges $\cup \{x \xrightarrow{c} y\}$
            endif
        enddo
    endif
enddo
return new_edges
```

Method: The tree traversal is started at the root vertex. Every time a link $u \xrightarrow{c} v$ from a partition P_τ to an other partition P_μ is traversed, we include u (and the preconditions c' which define how to reach u) in a set B of consecutively numbered "border objects" (these are vertices which have at least one edge to a vertex that belongs to a different partition). If B already contains a border object $b \in B | f(b) = f(u)$ – this is true if the path from the root of G^c down to u contains elements which are stored at the same server as u but in partitions different from P_τ – we substitute b with u. After that the server hosting P_μ is instructed to initialize its OLL catalog concerning P_μ regarding B. Here the first step is to create OLL catalog entries, i. e. we loop over $b \in B$ descendingly ordered by their sequence numbers, and create links from these b to v with respect to the precondition of b.[3] This loop stops if either all border objects are processed or a border object in scope is stored at the same server as P_μ. Thereafter the algorithm proceeds as described for the root server.

If a server finishes its OLL catalog initialization, it returns the created edges to its caller. This server copies all edges $u \overset{c}{\rightharpoonup} v$ where u is stored locally to its own OLL catalog. The remaining edges are added to the set of self-created OLL catalog entries and returned to its caller in turn. The procedure stops when the root server finishes.

5.2 OLL Catalog Extension

Product structures are frequently updated during the product development process. In most cases they do not affect the distribution of data. Object migration occurs rather seldom, as typically all partners and suppliers are interested in managing the data concerning their work share locally. Sometimes, however, product structures are not completely defined at startup of a project, so additional suppliers may have to be integrated into the development framework later. In this case updates of the OLL catalog have to be performed.

Algorithm 2 shows a fragment of a structure update procedure that calls the Update-procedure (algorithm 3) regarding the servers the newly connected objects are stored at.

Algorithm 2: (Extension of product structure)

```
CreateNewEdge(in vertex u, in vertex v, in condition c)
1   ...
2   if $(f(u) == f(v))$ then
3       dset ← Update^{(f(u))}$(u, \langle v, c\rangle, c, \emptyset)$
4   else
5       dset ← Update^{(f(u))}$(u, \langle v, \varepsilon\rangle, c, \{\langle u, c, 0\rangle\})$
6   endif
7   ...
```

Algorithm 3: (Extension of OLL Catalog)

```
Update(in vertex x, in ⟨vertex u, condition precond⟩,
          in condition $\bar{c}$, in borders B)
1   new_edges ← $\emptyset$
```

[3]The border object u has not to be regarded since $u \overset{c}{\rightharpoonup} v \in E$ (storing this edge in the OLL catalog would not lead to any additional information).

```
**if** $(B == \emptyset)$ **then**
    *seq* $\leftarrow 0$
**else**
    *seq* $\leftarrow min(B[z.seq\_number]) - 1$
**endif**
**if** $(\exists\ w \xrightarrow{c} x \in E)$ **then**
    **for all** $w \xrightarrow{c} x \in E$ **do**
        *pred.push* $\langle w \xrightarrow{c} x, \bar{c}\rangle$
    **enddo**
**else**
    *pred_edges* $\leftarrow$ *Init*$^{(f(u))}(u, precond, B)$
    **for all** $u \xrightarrow{c'} v \in$ *pred_edges* **do**
        **if** $(f(u) == f(x))$ **then**
            $OLL \leftarrow OLL \cup \{u \xrightarrow{c'} v\}$
        **else**
            *new_edges* $\leftarrow$ *new_edges* $\cup \{u \xrightarrow{c'} v\}$
        **endif**
    **enddo**
**endif**
**while** $(pred.test\,! = \mathrm{NULL})$ **do**
    $\langle x \xrightarrow{c} y, cond\rangle \leftarrow pred.pop$
    **if** $(f(x) == f(y))$ **then**
        **if** $(\exists\ w \xrightarrow{c''} x \in E)$ **then**
            **for all** $w \xrightarrow{c''} x \in E$ **do**
                *pred.push* $\langle w \xrightarrow{c''} x, cond \wedge c\rangle$
                **if** $(seq == 0)$ **then**
                    *precond* $\leftarrow$ *precond* $\wedge\ c$
                **endif**
            **enddo**
        **else**
            *pred_edges* $\leftarrow$ *Init*$^{(f(u))}(u, precond, B)$
            **for all** $u \xrightarrow{c} v \in$ *pred_edges* **do**
                **if** $(f(u) == f(x))$ **then**
                    $OLL \leftarrow OLL \cup \{u \xrightarrow{c} v\}$
                **else**
                    *new_edges* $\leftarrow$ *new_edges* $\cup \{u \xrightarrow{c} v\}$
                **endif**
            **enddo**
        **endif**
    **else**
        **if** $(\neg\exists b \in B | f(x) == f(b))$ **then**
            *pred_edges* $\leftarrow$ *Update*$^{(f(x))}(x, \langle u, precond\rangle,$
                            $cond \wedge c, B \cup \{\langle x, cond \wedge c, seq\rangle\})$
        **else**
            *pred_edges* $\leftarrow$ *Update*$^{(f(x))}(x, \langle u, precond\rangle,$
                                  $cond \wedge c, B)$
        **endif**
        **for all** $u \xrightarrow{c} v \in$ *pred_edges* **do**
            **if** $(f(u) == f(y))$ **then**
                $OLL \leftarrow OLL \cup \{u \xrightarrow{c} v\}$
            **else**
                *new_edges* $\leftarrow$ *new_edges* $\cup \{u \xrightarrow{c} v\}$
            **endif**
        **enddo**
    **endif**
**enddo**
**return** *new_edges*
```

Method: The general idea behind algorithm 3 is as follows:
Assume $u \xrightarrow{c} v$ to be the new edge. If we knew about all border objects in the path from the root object down to u we could simply call the Init-procedure (cf. algorithm 1) for u. To obtain information about all relevant border objects is the main task of algorithm 3. The procedure is to go backwards in the structure from u up to the root object and to collect all border objects and the conditions along the path. When the root object is reached the Init-procedure can be called.

Returning from the recursive descend each server includes the relevant new OLL edges – created by Init – in its local OLL catalog, the remaining edges are returned to the calling server.

5.3 OLL Catalog Usage

When expanding (or assembling) product structures, the usage of OLL catalogs is rather simple. We only have to distinguish between the "master" server (i. e. the server that stores the root vertex of the requested structure) and the other ones, the "slaves". The client which requests the structure initiates the multi-level expand by calling XMultiLevelExpand (cf. algorithm 4) which in turn calls the masterMLE and slaveMLE procedures (algorithms 5 and 6 respectively).

Algorithm 4: (Product structure expansion)

```
**XMultiLevelExpand(in node u, in assignment $\mathcal{A}$)**
*subgraph* $\leftarrow (\emptyset, \emptyset)$
*remotenodes* $\leftarrow \emptyset$
*masterMLE*$^{(f(u))}$*(u,$\mathcal{A}$,subgraph,remotenodes)*
**for all** $s \in S$ **in parallel do**
    $R_s = \{v \in$ *remotenodes* $: f(v) = s\}$
    *subgraph*$_s$ $\leftarrow (\emptyset, \emptyset)$
    **if** $(\neg(R_s == \emptyset))$ **then**
        *slaveMLE*$^{(s)}(R_s, \mathcal{A}, subgraph_s)$
        *Merge*$(subgraph, subgraph_s)$
```

```
10          endif
11     enddo
12     return subgraph
```

Method: XMultiLevelExpand is called with the vertex u to expand, and the assignment $\mathcal{A}$ of structure options the user selected. First the local structure is expanded (masterMLE), resulting in (1) a preliminary subgraph and (2) all nodes on remote servers which have to be expanded subsequently. The subtrees of all remote servers are requested in parallel (cf. lines 4 – 11). The Merge procedure (line 9) simply creates the union of all resulting subgraphs (i. e. $V = \bigcup_{s \in S} \tilde{V_s}, E = \bigcup_{s \in S} \tilde{E_s}$ where $(\tilde{V_s}, \tilde{E_s})$ is the resulting subgraph of server s).

Algorithm 5: (Expansion according to the OLL catalog (master))

```
masterMLE(in node u, in assignment $\mathcal{A}$,
          out ($\tilde{V_s}, \tilde{E_s}$), out nodes remotenodes)
1      $\tilde{V_s} \leftarrow \{u\}$
2      $\tilde{E_s} \leftarrow \emptyset$
3      successors $\leftarrow \{u\}$
4      for all $x \in$ successors do
5          successors $\leftarrow$ successors $\setminus \{x\}$
6          for all $x \xrightarrow{c} y \in E \cup OLL|\mathcal{A}(c)$ do
7              if $x \xrightarrow{c} y \in E$ then
8                  $\tilde{E_s} \leftarrow \tilde{E_s} \cup \{x \xrightarrow{c} y\}$
9              endif
10             if $(f(x) == f(y))$ then
11                 $\tilde{V_s} \leftarrow \tilde{V_s} \cup \{y\}$
12                 if $(\neg$marked_visited$(y))$ then
13                     $successors \leftarrow successors \cup \{y\}$
14                 endif
15             else
16                 remotenodes $\leftarrow remotenodes \cup \{y\}$
17             endif
18         enddo
19         mark_visited$(x)$
20     enddo
21     return
```

Method: masterMLE is called to assemble vertex u considering the assignment $\mathcal{A}$. The two nested loops in line 4 and 6 traverse the local substructure. In lines 7 – 9 the local edges are collected (OLL edges are not included in the result!). Local vertices are then added to the result, remote nodes are collected separately (lines 10 – 17). Visited vertices are marked in order to avoid multiple processing (line 19). The result of masterMLE consists of a) the local subgraph and b) a set of all directly or indirectly referenced vertices stored at remote sites.

Algorithm 6: (local expansion of a PS according to local OLL catalog entries (slave))

```
slaveMLE(in R_s, in assignment 𝒜, out (Ṽ_s, Ẽ_s))
1   $\tilde{V_s} \leftarrow R_s$
2   $\tilde{E_s} \leftarrow \emptyset$
3   $\textit{successors} \leftarrow R_s$
4   for all $x \in \textit{successors}$ do
5       $\textit{successors} \leftarrow \textit{successors} \setminus \{x\}$
6       for all $x \xrightarrow{c} y \in E \cup OLL|\mathcal{A}(c)$ do
7           if $x \xrightarrow{c} y \in E$ then
8               $\tilde{E_s} \leftarrow \tilde{E_s} \cup \{x \xrightarrow{c} y\}$
9           endif
10          if $(f(x) == f(y))$ then
11              $\tilde{V_s} \leftarrow \tilde{V_s} \cup \{y\}$
12              if $(\neg \textit{marked\_visited}(y))$ then
13                  $successors \leftarrow successors \cup \{y\}$
14              endif
15          endif
16      enddo
17      $\textit{mark\_visited}(x)$
18  enddo
19  return
```

Method: In contrast to the masterMLE procedure, slaveMLE is called with a *set* of vertices which are the root objects of local substructures. The rest of slaveMLE works similar to masterMLE, except that no remote objects have to be collected.

6 Discussion

During the past few years a lot of research work has been performed in the context of distributed query processing in all kinds of databases (cf. [Gra93], [HMS92], [Kos00], and [YM98]). The very special but important aspect of assembling distributed objects, however, has received little attention.

In [KGM91] and [MGS$^+$94] an "assembly" operator is introduced for combining distributed complex objects causing only a small number of communications to remote sites by doing as much work as possible at each location which is involved in the user's request: The remote sites assemble the object status locally as far as possible and return partial objects which have to be checked for further unresolved references. If any, the missing parts are retrieved from the corresponding servers. This is similar to the approach described in section 1, where a sequence of recursive SQL queries is executed to assemble the entire product structure. The problems are similar as well: Remote sites may be contacted more than once, and parts occurring more than once within the structure are retrieved multiply.

An evaluation strategy for executing recursive queries in distributed deductive databases is described in [NCW93]. The paper focuses on minimizing the distribution costs based on the idea of semi-joins in conventional databases. Avoiding multiple calls to a remote site is not in the scope of the approach presented there.

Initialization and usage of OLL catalogs are closely related to distributed or parallel execution strategies for transitive closure (cf. [CCH93], [HAC90]). The distributed product structure in our scenario may be seen as some kind of *semantic fragmentation* of the data according to the workshare defined by collaborating partners and suppliers. [HAC90] describes an approach similar to the one presented here: First, fragments which are involved in the query (based on the transitive closure) are determined by interpreting some sort of "fragment connection graph". Second, the – probably slightly modified – query is executed at each fragment in parallel. At last the partial results are combined according to the initial query. This method works well for e.g. the connection and shortest path problems. However, because of the quite imprecise connection information at the fragment level, the execution of multi-level expands using this approach would result in too many objects in the intermediate results which would have to be filtered out in the final "merging" step. To avoid this we store the precise information about the connected *objects* of different fragments within the OLL catalog.

At first glance, the OLL catalogs may look like the routing tables of computer networks ([Tan89]). Routing tables are used to route network packets between two nodes via the cheapest or shortest path. In contrast to our OLL catalog initialization, algorithms computing such tables typically use some sort of shortest path approaches.

OLL catalogs may bee seen as an index along path expressions, like a nested or path index (cf. [YM98]). Path indexes support the efficient evaluation of conditions in WHERE-clauses of queries. The benefit of path indexes is based on the fact that the evaluation of conditions in WHERE-clauses of queries does not need to traverse substructures of requested objects. In our case, however, queries such as the multi-level expand are to return an entire substructure by traversing it. In order to speed up this different kind of queries, we need an other type of index. OLL catalog entries do not depend on *values* of object attributes – in contrast to nested or path indexes – but on the *location* of the objects.

In this paper we regard product structures as a special kind of directed acyclic graphs. All algorithms presented here are specialized versions of tree traversals using the depth-first search (cf. [CLR96], [Sed88]).

Partitioning directed acyclic condition graphs seems to be similar to the derived horizontal partitioning of tables in distributed databases ([CP84]). However, efficient query processing in this special context has also received little attention only.

Analytical computations (cf. [MDEF01]) have shown that in most cases (assuming realistic product structures) the algorithms described in section 5 may eliminate up to 95% of the original response times of multi-level expand actions! Figure 4 illustrates the typical results of such computations: The response times of multi-level expand actions using the simple "traditional" approach are compared to those using the OLL approach on the basis of three differently sized product structures in three different network environments (the assumed latency times T_{lat} and data transfer rates dtr of configurations 1, 2 and 3 are $T_{lat} = 150ms$ and $dtr = 256kBit/s$, $T_{lat} = 150ms$ and $dtr = 512kBit/s$, and $T_{lat} = 50ms$ and $dtr = 1MBit/s$ respectively).

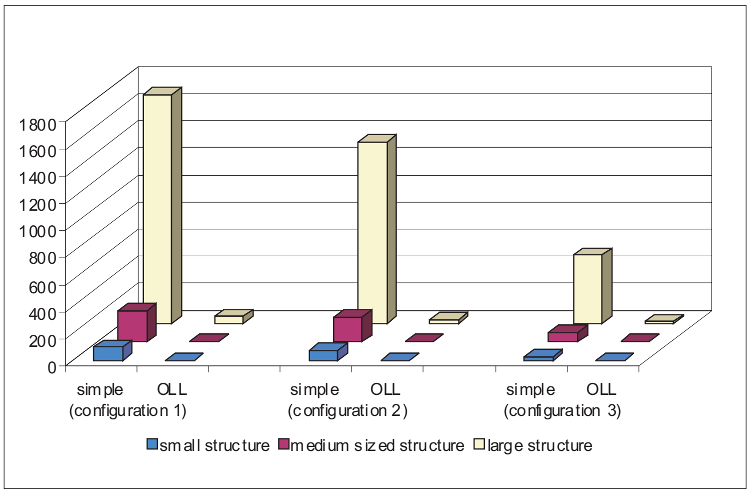

Figure 4: Multi-level expand response times of three different product structures in three different network environments

The OLL approach has been implemented experimentally using IBM DB2 UDB v7.2 as the underlying DBMS. The measurements performed showed response times which came very close to the analytically estimated ones.

7 Summary and Outlook

In order to survive severe competition, collaborative product development spanning several geographically distributed partners is becoming an indispensible must for manufacturing industries more and more. Even in wide area networks, fast access to the partners' data – especially for assembling product structures – is essential for efficient collaboration.

In this paper we introduced OLL catalogs. In short, they store information about transitive dependencies of parts, which are stored in different partitions at possibly different database servers. With these catalogs we are able to minimize the number of communications between involved database servers: Each server is contacted at most once, and even parts that occur more than once within the product structure are queried only once. As no intermediate results are necessary to generate queries that retrieve missing remote substructures, all queries can be executed in parallel, thus further minimizing the response times of assembly operations.

The OLL approach described in this paper shows the direction into which PDM systems have to move in order to meet the performance requirements of worldwide distributed product development.

References

[ANS99] ANSI/ISO/IEC 9075-2:1999 (E). *Database Language SQL – Part 2: Foundation (SQL/Foundation)*, September 1999.

[CCH93] Filippo Cacace, Stefano Ceri, and Maurice A. W. Houtsma. A Survey of Parallel Execution Strategies for Transitive Closure and Logic Programs. *Distributed and Parallel Databases*, 1(4):337–382, 1993.

[CIM97] CIMdata, Inc., CIMdata World Headquarters, Ann Arbor, MI 48108 USA. *Product Data Management: The Definition. An Introduction to Concepts, Benefits, and Terminology*, fourth edition, September 1997.

[CLR96] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT-Press, 1996.

[CP84] S. Ceri and G. Pelagatti. *Distributed Databases. Principles and Systems*. McGraw-Hill, 1984.

[EM99] A. Eisenberg and J. Melton. SQL:1999, formerly known as SQL3. *ACM SIGMOD Record*, 28(1):131–138, March 1999.

[Fit90] M. C. Fitting. *First-order logic and automated theorem proving*. Springer, New York, Heidelberg, 1990.

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[HAC90] M. Houtsma, P. Apers, and S. Ceri. Distributed Transitive Closure Computations: The Disconnection Set Approach. In *Proceedings of the 16th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Brisbane*, 1990.

[HMS92] T. Härder, B. Mitschang, and H. Schöning. Query Processing for Complex Objects. *Data & Knowledge Engineering*, 7:181–200, 1992.

[IBM01] IBM Corporation. *IBM DB2 Universal Database – SQL Reference – Version 7*, 2001.

[KGM91] T. Keller, G. Graefe, and D. Maier. Efficient Assembly of Complex Objects. In *Proceedings of ACM Sigmod Conference, Denver, Colorado, May 1991*, volume 20, pages 148–157. ACM Press, June 1991.

[Kos00] D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, September 2000.

[LE78] A. H. Lightstone and H. B. Enderton. *Mathematical logic: An introduction to model theory*. Plenum Pr. New York, 1978.

[MDEF01] E. Müller, P. Dadam, J. Enderle, and M. Feltes. Tuning an SQL-Based PDM System in a Worldwide Client/Server Environment. In *Proceedings of 17th International Conference on Data Engineering, Heidelberg, Germany*, pages 99–108, 2001.

[MGS$^+$94] D. Maier, G. Graefe, L. Shapiro, S. Daniels, T. Keller, and B. Vance. Issues in Distributed Object Assembly. In M. T. Özsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management (Proceedings of the 1992 International Workshop on Distributed Object Management, August 1992, Edmonton, Canada)*, pages 165–181. Morgan Kaufmann Publishers, 1994.

[MOn] Matrix One. www.matrix-one.com.

[MP] Metaphase. http://www.plmsol-eds.com/metaphase/index.shtml.

[NCW93] W. Nejdl, S. Ceri, and G. Wiederhold. Evaluating Recursive Queries in Distributed Databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(1):104–121, February 1993.

[OLR95] A. Obank, P. Leaney, and S. Roberts. Data mangement within a manufacturing organization. *Integrated Manufacturing Systems*, 6(3):37–43, 1995.

[Sed88] R. Sedgewick. *Algorithms*. Addison Wesley, 2nd edition, 1988.

[Tan89] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall International Editions, second edition, 1989.

[WT] Windchill Technologies. http://www.ptc.com/products/windchill/index.htm.en.

[YM98] C. T. Yu and W. Meng. *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann Publishers, San Francisco, California, 1998.