Tuning an SQL-Based PDM System in a Worldwide Client/Server Environment

E. Müller, P. Dadam, J. Enderle
University of Ulm
Faculty of Computer Science
{*mueller,dadam,jost.enderle*}*@informatik.uni-ulm.de*

M. Feltes
DaimlerChrysler
Research and Technology Ulm
michael.feltes@daimlerchrysler.com

Abstract

The management of product-related data in a uniform and consistent way is a big challenge for many manufacturing enterprises, especially the large ones like DaimlerChrysler. So-called Product Data Management systems (PDMS) are a promising way to achieve this goal. For various reasons PDMS often sit on-top of a relational DBMS using it (more or less) as a simple record manager. User interactions with the PDMS are translated into series of SQL queries. This does not cause too much harm when DBMS and PDMS are located in the same local-area network with high bandwidth and little latency times. The picture may change dramatically, however, if the users are working in geographically distributed environments. Response times may rise by orders of magnitude, e. g. from 1-2 minutes in the local context to 30 minutes and even more in the "intercontinental" context. The paper shows how a more sophisticated utilization of the (advanced) SQL features coming along with SQL:1999 can help to cut down response times significantly.

1. Introduction – The Application Scenario

Product development is a time-consuming and costly process. Keen competition forces the companies to shorten this process more and more in order to survive. During the last years enormous endeavours have been made to optimize the disciplines involved in the engineering process. Most development departments for example introduced CAD (**C**omputer **A**ided **D**esign) and CAE (**C**omputer **A**ided **E**ngineering) tools leading to remarkable increases in productivity and significant reductions in time to market. But for all that, such *intra*-disciplinary, specialized tools are not able to support the engineering process as a whole and, therefore, limit further improvements to single sections of the overall process. The limitations result from poor capabilities for searching data efficiently, missing mechanisms to preserve the correctness and consistency of shared data, scarce provision of a uniform change and configuration management, and last but not least insufficient support of working in parallel. So *inter*-disciplinary optimizations became an indispensable must. – The idea of *Product Data Management* (PDM) was born.

The philosophy behind PDM systems addresses two important functions in a manufacturing company: To manage the enormous amount of information defining a product and to control the processes employed to manage the evolution of a product from the early stages of conception and design through to after sales and maintenance (cf. [1], [5], [13]).

Typically, a product has a recursively defined hierarchical structure. It is composed of assemblies and single parts (so-called components). All objects of that so-called product structure may be described by specifications, CAD files, work orders, simulation results, and much more. From this point of view a product is a very complex object.

In the different stages of product development, different users with various skills and tasks need to access the product data. In order to perform their tasks, the users often need different views on the product: Designers are mostly interested in shapes and surfaces, engineers need the physical structure of a product, and users responsible for functional modules need to see the same product decomposed into its functional units.

Hence, a PDM system has to solve two critical problems: At first, the large amount of data, forming a complex object structure, and second the different views on this structure. A common solution to this is to store the data using standard relational database systems. Therefore, the object structure is flattened, and all objects – and the relations between them, too – are stored in (more or less) ordinary, normalized tables in the database system. At runtime, when a user accesses a product in his/her view the corresponding structure information and data items are retrieved, interpreted, and reassembled.[1]

[1]This "flat" object respresentation looks very strange at first glance. Using extended attribute types of object-relational DBMS instead seems to be much better suited. But the product structure is (a) a recursive one and (b) different hierarchical views may have to be supported in parallel on the same set of data. Thus hierarchically structured complex objects – as offered by some object-relational DBMS – do not help. As (c) there exist efficient implementations for the processing of recursive SQL queries [10] meanwhile, the "neutral" flat representation makes in fact even sense.

A very typical way of using a PDM system is to navigate through a product structure. The users start with a product (the top level item of the product structure) and expand the next level of the structure. They repeat this so-called *single-level expand* until they find what they look for, or the branches of interest within the structure are expanded completely. In doing so, SQL is used as a simple record (or tuple) manager: The navigational traversal of the product tree is translated nearly one-to-one into single, isolated SQL queries. This stepwise ”navigational access” also works for the so-called *multi-level expand* which expands the entire object structure by recursively applying the single-level expand method. This leads to a large number of SQL queries.

Nevertheless, there is hardly any problem with this procedure in local-area networks (LANs). Because of the typically large data transfer rates and the very low latency times in a LAN environment, acceptable response times can be achieved. This picture changes dramatically, however, when applying the same procedure to worldwide distributed application environments. Response times may rise to an extent which is far beyond that what users are willing to accept.

Driven by some experiences in prototypical but realistic PDM environments at DaimlerChrysler we were looking for mechanisms to optimize such PDM systems without questioning their entire system architecture. Our suspicion was that the problem is caused by the large number of isolated queries in conjunction with lately evaluated user access rules resulting in many messages and a large amount of data to be transferred. This seems not to be a PDM-specific problem. Similar experiences have been made within SAP R/3 (cf. [6]), distributed databases (cf. [4]), and other client/server applications that use the data-shipping strategy (cf. [9]). The question arises if a more function-shipping oriented strategy would help and how an adequate solution could look like.

Two approaches utilizing the existing power of relational database systems and the new features introduced by the lately issued standard SQL:1999 (cf. [2], [8]) seemed to be very promising regarding our performance problem: The first approach attempts to reduce the amount of transferred data by early evaluation of access rules. The second approach takes advantage of the power of recursive queries which can reduce the number of queries – and hence the number of communications – significantly. But before doing any implementations like customizations or prototyping we were interested in the improvements that potentially might result from these database related ”tuning actions” in order to decide, whether their realization is worth its – possibly high – implementation costs or not. The results of these investigations are presented in this paper.

The rest of this paper is organized as follows: Section 2 describes the formulas we are using to compute expected response times. Different types of rules and conditions typically used in PDM systems are discussed in section 3. In sections 4 and 5 two approaches for optimizing the response times are discussed. The results are sumarized and rated in section 6. Section 7 finishes with a summary and an outlook on further work.

2. The Response Time Problem

As already mentioned, response times of user actions can become extremely long because of limited bandwidth and long latency times in a wide-area network (WAN). This is true especially for actions like so-called multi-level expands which typically retrieve a larger (sub-)tree of the complete object structure. In a testing environment for example such a multi-level expand was finished after only little more than half a minute using the LAN, whereas the same operation took up to half an hour using the WAN.

In order to find out the parameters worth for optimization we first cast a short look at the computation (in the sense of prediction) of response times from the view of database accesses[2]. Table 1 lists some definitions we will use in the following.

Table 1. Definitions for the computation of response times

symbol	description
dtr	data transfer rate in the WAN
T_{Lat}	latency time in the WAN
$size_p$	packet size in the WAN
$\not{\! o}size_n$	average size of a node in the object tree
$n_v(t)$	number of visible nodes[3] in a subtree t
$n_t(t)$	number of transmitted nodes of a subtree t
q	number of necessary database queries
c	number of necessary WAN-communications
vol	data volume resulting from user action
T	response time

To simplify the computation we assume that each query can be transmitted by using only *one* message (packet). Using the definitions in table 1 the response time for an action that retrieves a product structure tree can be computed as follows:

Because of the navigational access each node is touched and its data and the references to its subtrees or leaf nodes are fetched. This leads to as many queries q_s as there are nodes in the tree visible to the user (s stands for ”simple”

[2]In the following we use the term *response time* of a user action as a synonym for the *”accumulated delay caused by database accesses through a wide area network”*.

[3]The user may not be allowed to see all nodes in the tree (for details see section 3).

navigational process):

$$q_s = n_v(t) \tag{1}$$

Since every query causes an answer there are twice as many communications c_s as queries, so

$$c_s = 2 * q_s \tag{2}$$

The data volume vol_s that has to be transmitted is the sum of the transferred query data and the corresponding responses. To be more precise, in the average we expect the last package of each response to be filled only half. In order to take this fact into account we add a correcting term. The resulting data volume can then be approximated by

$$vol_s = q_s * size_p + n_t(t) * \not{\phi} size_n + q_s * \frac{1}{2} size_p \tag{3}$$

By combining the equations (2) and (3) we achieve the overall response time:

$$T_s = c_s * T_{Lat} + vol_s / dtr \tag{4}$$

In table 2 the results of some computations considering queries, single-level expands, and multi-level expands are listed. In all examples we have assumed a complete ν-ary tree (i. e. all leaves have the same depth and all internal nodes have degree ν). The parameters τ and ν refer to the depth of a tree and the number of branches each node has. The figure σ refers to the probability that a user is allowed to see a branch. This is an estimation of the effects of the rules described in the subsequent section. In a ν-ary tree the number of visible nodes (cf. equation (1)) and the number of transmitted nodes (cf. equation (3)) can be computed as follows[4]:

$$\begin{aligned} n_v(t) &= \sum_{i=1}^{\tau} (\sigma * \nu)^i \\ n_t(t) &= \begin{cases} \sum_{i=1}^{\tau} \nu^i & \text{for queries} \\ \nu & \text{for single-level expands} \\ \nu * \sum_{i=0}^{\tau-1} (\sigma * \nu)^i & \text{for multi-level expands} \end{cases} \end{aligned}$$

A ”query” is assumed to retrieve all nodes of a tree (without the structure information), a ”single-level expand” retrieves only the direct children of the root, and the ”multi-level expand” retrieves the entire structure[5]. The computation results in table 2 show that – depending on the environment and the object structure – queries and multi-level expands result in response times from several seconds up to nearly half an hour – and frustrated users, too!

In order to achieve acceptable response times we will focus on minimizing the data volume by early evaluation of rules and conditions (see section 3) as well as minimizing the number of executed queries.

[4]The root object is considered to be already at the client and therefore is not taken into account here.

[5]The multi-level expand uses a recursive approach: The single-level expand is applied to the root object, the resulting objects are filtered according to the rules, and the ”surviving” objects are then expanded recursively.

3. Rules and Conditions in PDM Systems

3.1. Types of Rules

Before giving a classification of rules we will describe the different types of rules typically used in PDM systems.

The first kind of rules is used to control the configuration of a product. Such so-called *structure options* are evaluated for controlling alternative or supplementary parts of a product. Consequently, an object associated with a structure option is part of the current product version only if the user has specified at least that structure option. As a result, structure options have to be evaluated when accessing the structure of the current product version.

Obviously, during the configuration process not every combination of the offered features is valid. For example it is not possible to choose a cabriolet together with a sunroof. Such dependencies between structure options are handled by so-called *configuration rules*. In contrast to the evaluation of structure options, configuration rules can be evaluated by accessing the selected structure options only. No access to additional data is necessary, in particular no product data need to be retrieved from the database. Therefore we will not look at optimization possibilities of configuration rules in the rest of this paper.

Effectivities, another type of rules, are very similar to structure options. They both are usually associated with relations between objects that require additional management based on either dates or unit numbers of parts. Effectivities are very useful to control product structures containing objects only available during a limited time or production period. So objects are included in a current product only if the associated effectivity overlaps the effectivity selected by the user.

The last kind of rules we want to look at are the *(message) access rules*. Not every user is allowed to perform each operation on – in other words: to send each message to – an arbitrary object. Object access has to be limited therefore. As we are only interested in messages involving database access we focus on messages like *multi-level expand* (object tree expansion) and *check-out/check-in* (gaining exclusive access to an object for updates).

In general message access rules are applied either to permit or to prohibit access to data under certain conditions[6]. We can conceive those rules as 4-tupels: A *user* is permitted to perform an *action* on an instance of an *object type*, if the *condition* is met.

[6]In the following we assume that rules only *permit* users to perform certain actions (that is, we assume to have a negative biased rule processing system). This is no real restriction because the positive biased rule processing system can be achieved by negating the conditions.

Table 2. Response times for several scenarios in today's environments

$size_{packet} = 4\text{kB}$	$\tau = 3, \nu = 9, \sigma = 0.6$			$\tau = 9, \nu = 3, \sigma = 0.6$			$\tau = 7, \nu = 5, \sigma = 0.6$		
$\not{\phi} size_{node} = 512\text{Byte}$	Query	Exp	MLE	Query	Exp	MLE	Query	Exp	MLE
T_{Lat} = 0.15	0.30	0.30	57.91	0.30	0.30	133.52	0.30	0.30	984.00
dtr = 256	12.98	0.33	41.19	461.48	0.23	95.01	1526.05	0.27	700.39
$T_s = \Sigma$	13.28	0.63	99.10	461.78	0.53	228.53	1526.35	0.57	1684.39
T_{Lat} = 0.15	0.30	0.30	57.91	0.30	0.30	133.52	0.30	0.30	984.00
dtr = 512	6.49	0.16	20.60	230.74	0.12	47.51	763.02	0.13	350.20
$T_s = \Sigma$	6.79	0.46	78.50	231.04	0.42	181.02	763.32	0.43	1334.20
T_{Lat} = 0.05	0.10	0.10	19.30	0.10	0.10	44.51	0.10	0.10	328.00
dtr = 1024	3.25	0.08	10.30	115.37	0.06	23.75	381.51	0.07	175.10
$T_s = \Sigma$	3.35	0.18	29.60	115.47	0.16	68.26	381.61	0.17	503.10

Data transfer rate *dtr* in kBits per second, latency time T_{lat} and response time T_s in seconds;
response times are split into the two parts caused by the latency time and the data transfer

Examples:

1. user: *Scott*
   action: *multi-level expand*
   type: *assembly*
   cond: $assembly.make\_or\_buy \neq 'buy'$
   permits user Scott to perform a multi-level expand on an instance of "assembly" if it is not bought from a supplier.
2. user: *
   action: *check-out*
   type: *tree(assembly)*
   cond: $\forall n \in tree(assembly) : n.checkedout \neq TRUE$
   permits every user to check-out an entire subtree (which root is of type "assembly") if all nodes *n* in this subtree are checked-in.

For evaluation purposes it would be desireable to represent the structure options and effectivities in the same way the message access rules are represented. And, indeed, this can be achieved rather easily: We stated that structure options and effectivities are associated with relations between objects. If we regard these relations as "first class" objects, we can formulate the rules for structure options (and effectivities) as follows:

3. user: *
   action: *access*
   type: *relation*
   cond: *relation.strc_opt* overlaps *user_strc_opt*
   permits every user to access (traverse) the relation if the set of structure options associated with this relation overlaps the user-selected ones.

So, in the following we will handle these conditions exactly in the same way we handle the message access rules.

3.2. Classification of Conditions

After discussing the different types of rules in section 3.1 we will now turn towards the conditions which may occur in the rules. Figure 1 shows a classification tree of the conditions we must be able to handle.

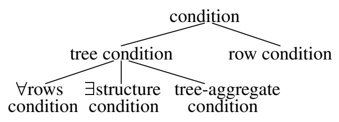

Figure 1. Classification of conditions

First of all, we distinguish between *tree conditions* and *row conditions*. The former describes a rule involving the whole object tree, whereas the latter involves only one simple object within a tree (typically the root object). Example 1 in section 3.1 uses a row condition, whereas example 2 uses a tree condition.

The row conditions can be very simple ones that can be evaluated by the use of standard SQL predicates using the conventional comparison operators ($<, >, \leq, \ldots$). If these SQL predicates are not sufficient to evaluate the condition, like the comparisons of sets or intervals, stored functions ([3], [7], [12]) performing the checks have to be provided at the server.

Tree conditions can be split into three subclasses: The first includes all conditions which use the "for-all" quantor (cf. example 2 in section 3.1). All nodes in the tree have to meet the given condition which itself is a row condition. We will call these conditions "$\forall$rows conditions".

The second subclass which we call "$\exists$structure conditions" contains all conditions which refer to related objects of the tested object. For example, the state of a component *cmp* (i. e. a single part) may be frozen (i. e. unchangeable in the future) only if there exists a specification related to that component. Those conditions are written as "$\exists s \in \text{SPEC} : cmp \xrightarrow{specified\text{-}by} s,\ cmp \in nodes(tree(root\_obj)) \cap components$", where SPEC is the class of specifications and *specified-by* represents the relation between components and specifications.

The last subclass of the tree conditions contains all conditions which include a tree aggregate or tree function. Those conditions cannot be evaluated at a single node (or object) because they involve the entire tree (for example the number of nodes in the tree or the average of an attribute common to all nodes in the tree). Those conditions may be written as e. g. "*count*(*tree*(*assy*)) $\leq$ 10" or "*average*(*tree*(*assy.weight*)) $\leq$ 12". We will call these conditions "tree-aggregate conditions".

In the next two sections we will analyze how these conditions and rules can be expressed using SQL, and how the queries used so far have to be modified accordingly. In simple cases the WHERE-clause will be extended by additional

”AND”-conditions, in other cases more complex modifications become necessary. To simplify the discussion we first focus on finding appropriate SQL predicates in an isolated fashion. The combination of such predicates with existing queries in order to create more powerful queries is treated in section 5.5.

4. Approach 1: Reducing the Transferred Data Volume by Early Rule Evaluation

In our environment expands e.g. for digital mockups need to retrieve the entire structure from the root down to each single leaf. Therefore special solutions like stopping the recursive descent if predicates that exploit the semantics of the hierarchy indicate the irrelevance of deeper levels (cf. [14]) do not work. We propose the reduction of data volume by early evaluation of access rules instead.

4.1. Query Modification for Early Rule Evaluation

Because of the navigational approach each query that is part of a tree request retrieves all (and only) the directly related (sub-)objects of one object. As only a small part of the tree is accessed one cannot evaluate arbitrary tree conditions within such a navigational query in general. Therefore we can restrict our discussion to the representation of row conditions and their evaluation at this point.

Row conditions are based on comparisons containing object attributes, constants, variables of the user’s environment and functions calculated upon these values. Obviously such conditions can be transformed straightforward into an SQL WHERE clause. The condition in example 1 could be embedded into an existing query as follows:

```
SELECT ... FROM ..., assembly
WHERE ... AND assembly.make_or_buy<>’buy’
```

Row conditions may also refer to so-called *transient attributes* which are computed by the PDM system. If this computation cannot be directly transformed into an equivalent SQL expression, a user-defined function performing the respective computation has to be provided at the database server.

This means that the row conditions can be transformed quite easy into equivalent SQL WHERE clauses. In order to minimize the transformation effort at runtime, it is appropriate to automatically transform the conditions only once into an equivalent SQL predicate directly after the definition of a new rule. The transformed representation can be stored together with the corresponding rule (cf. section 3.1) in a table at the client. By doing this the query modificator can determine which conditions apply (by simply accessing the table) and therefore which SQL clauses have to be integrated into the WHERE clause of the current query. Two or more qualifying conditions are always connected via the ”OR” operator, and the resulting predicate is either appended to an already existing WHERE clause with an ”AND” or a new WHERE clause has to be generated. Obviously, query modification is very simple for row conditions.

4.2. Maximal Improvement

As the number of queries did not change in section 4.1, the only improvement can be expected by the reduction of the retrieved data volume. Let σ $(0 \leq \sigma \leq 1)$ denote the selectivity of a rule with respect to a query q. Then $1 - \sigma$ denotes the share of objects returned by q but not visible to the user. Assume o_q to be the number of objects returned by the query q. Then the maximally achievable improvement will be $T_{diff} \approx \frac{(1-\sigma)*o_q*\phi size_{node}}{dtr}$. For our examples in table 2 this would lead to the improved results shown in table 3: Response times of query actions go down from nearly half an hour to approximately one minute or even less. In contrast, the benefit gained by a multi-level expand is very low. Only a few seconds can be saved, leaving response times still beyond the level users are willing to accept.

Conclusion: It is not enough to reduce the data volume by evaluating access rules within the queries. Obviously, the more critical point is to reduce the number of round trips to the database in order to save latency times. We will focus on this aspect in the next section.

5. Approach 2: Reduction of Round Trips to the Database

5.1. Method

In order to reduce the number of round trips to a database we must achieve a reduction of the number of transmitted SQL-calls necessary to perform the requested user action. In our application context, the compilation of previously isolated queries resulting from a certain tree-oriented user action – like a multi-level expand – into one combined query appears to be very promising. Here we will take advantage of the recursive structure of the object trees.

5.2. Utilization of Recursive SQL

In principle, with recursive SQL (as defined in the SQL:1999 standard, cf. [2] and [8]) we are able to collect all nodes of a recursively defined object tree in one query. However, *one query* implies *one result type*. This is no problem at least if all nodes in the tree are of the same type. But in general an object tree may consist of nodes of many different types, so the objects have to be unified regarding their type without loosing their object type information[7].

[7]First attempts to solve this problem in the context of inheritance can be found in the IBM DB2 UDB V6 [10] and in the Informix Dynamic Server.2000 Version 9.2 [11].

Table 3. Response times for several scenarios with early rule evaluation

$size_{packet} = 4$kB	$\tau = 3, \nu = 9, \sigma = 0.6$			$\tau = 9, \nu = 3, \sigma = 0.6$			$\tau = 7, \nu = 5, \sigma = 0.6$		
$\emptyset size_{node} = 512$Byte	Query	Exp	MLE	Query	Exp	MLE	Query	Exp	MLE
T_{Lat} = 0.15	0.30	0.30	57.91	0.30	0.30	133.52	0.30	0.30	984.00
dtr = 256	3.19	0.27	39.19	7.13	0.22	90.39	51.42	0.23	666.23
$T_s = \Sigma$	3.49	0.57	97.10	7.43	0.52	223.90	51.72	0.53	1650.23
saving in %	73.74	8.96	2.02	98.39	3.51	2.02	96.61	5.52	2.03
T_{Lat} = 0.15	0.30	0.30	57.91	0.30	0.30	133.52	0.30	0.30	984.00
dtr = 512	1.59	0.14	19.60	3.56	0.11	45.19	25.71	0.12	333.12
$T_s = \Sigma$	1.89	0.44	77.50	3.86	0.41	178.71	26.01	0.42	1317.12
saving in %	72.12	6.06	1.27	98.33	2.25	1.28	96.59	3.61	1.28
T_{Lat} = 0.05	0.10	0.10	19.30	0.10	0.10	44.51	0.10	0.10	328.00
dtr = 1024	0.80	0.07	9.80	1.78	0.05	22.60	12.86	0.06	166.56
$T_s = \Sigma$	0.90	0.17	29.10	1.88	0.15	67.10	12.96	0.16	494.56
saving in %	73.19	7.73	1.69	98.37	2.96	1.69	96.61	4.69	1.70

A feasible solution for this unification is to define a new (result-) type enfolding all attribute definitions of all object types appearing in the result plus an additional attribute – if it does not exist anyway – containing the original object type information. The attribute values of a resulting object can then be mapped to the corresponding attributes in the result type and the remaining attributes are filled with NULL values. – We will point out the basic idea with a little example.

assy	type	obid	name	dec
	assy	1	Assy1	+
	assy	2	Assy2	+
	assy	3	Assy3	+
	assy	4	Assy4	+
	assy	5	Assy5	−
	assy	6	Assy6	−
	assy	7	Assy7	−
	assy	8	Assy8	−

comp	type	obid	name
	comp	101	Comp1
	comp	102	Comp2
	comp	103	Comp3
	comp	104	Comp4
	comp	105	Comp5
	comp	106	Comp6
	comp	107	Comp7

link	type	obid	left	right	eff_from	eff_to
	link	1001	1	2	1	3
	link	1002	1	3	4	10
	link	1003	2	4	1	10
	link	1004	2	5	1	10
	link	1005	4	101	6	10
	link	1006	4	102	1	5
	link	1007	5	103	1	10
	link	1008	5	104	1	10

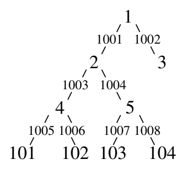

Figure 2. Tables for assemblies, components, and their relation, forming a tree

Figure 2 shows the tables for the assemblies, components, and the links between them. The relation named "assy" contains several assemblies, each of which has an object ID, a name and a flag indicating whether the assembly is decomposable without destroying it or not (see attribute "dec"). The relation called "comp" contains several single parts, each of which has an object ID and a name. The structural relationship between assemblies and components is stored in the relation "link". Each link refers with its attribute "left" to an assembly while "right" refers to a contained assembly or component. The attributes "eff_from" and "eff_to" contain the beginning and ending number of the effectivity (e.g. lot numbers) respectively.

The following recursive query retrieves the whole tree stored in the three tables of figure 2 and generates the "uniform" result table[8]:

```
WITH RECURSIVE rtbl (type, obid, name, dec) AS
(SELECT type, obid, name, dec
 FROM   assy
 WHERE  assy.obid = 1
    UNION
 SELECT assy.type, assy.obid, assy.name, assy.dec
 FROM   rtbl JOIN link ON rtbl.obid=link.left
        JOIN assy ON link.right=assy.obid
    UNION
 SELECT comp.type, comp.obid, comp.name, ''
 FROM   rtbl JOIN link ON rtbl.obid=link.left
        JOIN comp ON link.right=comp.obid
)
SELECT type, obid, name, dec AS "DEC",
       cast (NULL AS integer) AS "LEFT",
       cast (NULL AS integer) AS "RIGHT",
       cast (NULL AS integer) AS "EFF_FROM",
       cast (NULL AS integer) AS "EFF_TO"
FROM   rtbl
   UNION
SELECT type, obid, '' AS "NAME", '' AS "DEC",
       left, right, eff_from, eff_to
FROM   link
WHERE  (left IN (SELECT obid FROM rtbl)
  AND   right IN (SELECT obid FROM rtbl))
ORDER BY 1,2
```

In this query we ignored potentially existing access rules, structure options, and effectivities. The first part of the query beginning with the "`WITH...`" clause walks through the object tree and collects all contained assemblies and components. In the second part of the query all those parts are selected and casted to the result type. As this information is not sufficient to reconstruct the original object tree, the last part of the query retrieves all necessary link objects and casts them to the result type, too. – The result of this query is shown in figure 3.

Of course, if there exist access rules we do not want to transmit the entire object tree. This would cause unnecessary network traffic again and thus worsen the response time of the action. In order to achieve acceptable response times we must combine the rule evaluation with the recursive query statement in an analogous way we did in section 4. Row conditions can be handled as described in section 4.1. Therefore only the procedure of translating tree conditions and the appropriate modification of the recursive query are discussed in the next sections.

[8]All recursive queries in this paper can be evaluated with minor syntactical changes by IBM DB2 UDB V6.

TYPE	OBID	NAME	DEC	LEFT	RIGHT	EFF_FROM	EFF_TO
assy	1	Assy1	+	-	-	-	-
assy	2	Assy2	+	-	-	-	-
assy	3	Assy3	+	-	-	-	-
assy	4	Assy4	+	-	-	-	-
assy	5	Assy5	—	-	-	-	-
comp	101	Comp1	-	-	-	-	-
comp	102	Comp2	-	-	-	-	-
comp	103	Comp3	-	-	-	-	-
comp	104	Comp4	-	-	-	-	-
link	1001	-	-	1	2	1	3
link	1002	-	-	1	3	4	10
link	1003	-	-	2	4	1	10
link	1004	-	-	2	5	1	10
link	1005	-	-	4	101	6	10
link	1006	-	-	4	102	1	5
link	1007	-	-	5	103	1	10
link	1008	-	-	5	104	1	10

Figure 3. Result of recursive query without rule evaluation

5.3. Representation of Tree Conditions in SQL

In the following we analyze which kind of predicates can be mapped to which kind of query.

5.3.1. ∀Rows Conditions. ∀rows conditions are of the following form:

$\forall obj \in$ nodes(tree($root\_obj$)) : row_cond(obj)

where "nodes" is a set-valued function returning all nodes in a tree with root *root_obj*, and "row_cond" is a valid row condition which has to be met by all these nodes. This means that the resulting tree contains *all* nodes visible to the user if all nodes meet the *row_cond*. If at least one node does not meet this condition the result tree is empty!

Now the question arises how to transform such a condition into an SQL statement. Assume that *rec_table* contains all nodes of the tree without computing the ∀rows condition. Then we can implement the "all-or-nothing" principle as follows:

SELECT * FROM rec_table WHERE NOT EXISTS (
      SELECT * FROM rec_table WHERE NOT *row_cond*)

If the subselection retrieves at least one object that does not meet the *row_cond*, then the outer select returns no object. If the subselection does not find a match then the outer select returns all objects included in the *rec_table*. (Please note that *rec_table* occurs in the outer and in the inner clause! But an intelligent query optimizer will recognize that the inner clause needs to be evaluated only once, as it is an uncorrelated sub-query.)

We will show the effect of this clause by our example (see figure 2). Assume that all assemblies in the resulting tree have to be decomposable. The appropriate row condition for this is: $assy.dec =$ ' + '. If there exists at least one assembly in the tree which is not decomposable the result is empty. The following query achieves this:

```
WITH RECURSIVE rtbl (type, obid, name, dec) AS
( ... as in section 5.2 ...)
SELECT type, obid, name, dec AS "DEC",
       cast (NULL AS integer) AS "LEFT",
       cast (NULL AS integer) AS "RIGHT",
       cast (NULL AS integer) AS "EFF_FROM",
       cast (NULL AS integer) AS "EFF_TO"
FROM   rtbl
WHERE NOT EXISTS (SELECT * FROM rtbl
                  WHERE (type='assy' AND dec!='+'))
   UNION
SELECT type, obid, '' AS "NAME", '' AS "DEC",
       left, right, eff_from, eff_to
FROM   link
WHERE  (left IN (SELECT obid FROM rtbl)
  AND   right IN (SELECT obid FROM rtbl))
  AND   NOT EXISTS (SELECT * FROM rtbl
                    WHERE (type='assy' AND dec!='+'))
ORDER BY 1,2
```

The result of this query is empty because of assembly number five. It is not decomposable, so no tree will be returned.

5.3.2. ∃structure Conditions. As we showed in section 3.2 ∃structure conditions are of the following form:

$\exists u \in U : o \xrightarrow{rel} u,\ \ o \in$ nodes(tree($root\_obj$)) $\cap\, O$

This condition means that there must exist an object u of type U so that the tested object o of type O is related to that u via the relation *rel*. In order to decide whether there exists such an object u, we have to join the tables O, *rel*, and U. For being able to do so we need some additional information about *how* this join has to be performed. We assume that the objects o and u both can be identified by an attribute called "obid", and the relation between them refers to o and u with an attribute called "left" and "right" respectively. Then we can implement the ∃structure condition as follows:

SELECT * FROM O WHERE EXISTS (
SELECT * FROM rel JOIN U ON rel.right=U.obid
WHERE O.obid=rel.left)

Remark: The object o may be an arbitrary node within the tree. If the tree contains objects of different types, O need not necessarily be the type of the root object. As a result, although the ∃structure conditions are defined at the root object, they have to be evaluated at objects of type O!

Again, we will show the effect of this condition by an example. We extend our example from above by a relation called "spec" containing specification documents for assemblies and components, and a relation called "specified_by" that links the specifications to the objects. Then we want to query for all objects in the tree with the restriction that components are visible only if they are specified by at least one document ($\exists s \in spec : c \xrightarrow{specified\_by} s$, $c \in nodes(tree(1)) \cap comp$):

```
WITH RECURSIVE rtbl (type, obid, name, dec) AS
(SELECT type, obid, name, dec
 FROM   assy
 WHERE  assy.obid = 1
    UNION
 SELECT assy.type, assy.obid, assy.name, assy.dec
 FROM   rtbl JOIN link ON rtbl.obid=link.left
        JOIN assy ON link.right=assy.obid
    UNION
 SELECT comp.type, comp.obid, comp.name, ''
 FROM   rtbl JOIN link ON rtbl.obid=link.left
        JOIN comp ON link.right=comp.obid
 WHERE  EXISTS (SELECT * FROM specified_by AS s JOIN spec
        ON s.right = spec.obid WHERE s.left = comp.obid)
)
SELECT ... as in section 5.2 ...
```

5.3.3. Tree-Aggregate Conditions. Quite similar to the $\forall$rows conditions the tree-aggregate conditions exclude either the whole tree or nothing. Obviously the translation into SQL conditions will not look very differently.

Tree-aggregate conditions use aggregate functions like AVG, COUNT, MAX, MIN, and SUM which refer to the entire object tree. They may have the following form:

$agg\_func(o.x) \otimes expr, \ \ o \in \text{nodes}(\text{tree}(root\_obj))$

where *agg_func* is one of the aggregate functions, $\otimes$ is one of the common compare operators, and *expr* is an expression that may contain aggregate functions, constants and attribute values of the root object of the tree.

Since our recursive query retrieves all accessible nodes of the tree, we can evaluate the tree-aggregate functions on that set of nodes. Assume that *rec_table* is the result of the recursive query. The translation of such a tree-aggregate condition looks as follows:

SELECT * FROM rec_table WHERE (
SELECT *agg_func*(*attr*) FROM rec_table) $\otimes$ *expr*

If the aggregate function of the tree-aggregate condition should be evaluated only on a subset of all nodes in the tree, the WHERE clause of the subselection has to be adapted accordingly.

Finally we will show the effect of this condition by our example. The assumed user may only retrieve trees containing at most ten assemblies (again, the type-discriminating attribute in the homogenized result of the recursion must be used to identify tuples of the considered type):

```
WITH RECURSIVE rtbl (type, obid, name, dec) AS
(... as in section 5.2 ...)
SELECT type, obid, name, dec AS "DEC",
       cast (NULL AS integer) AS "LEFT",
       cast (NULL AS integer) AS "RIGHT",
       cast (NULL AS integer) AS "EFF_FROM",
       cast (NULL AS integer) AS "EFF_TO"
FROM   rtbl
WHERE (SELECT COUNT(*) FROM rtbl WHERE type='assy')<=10
   UNION
SELECT type, obid, '' AS "NAME", '' AS "DEC",
       left, right, eff_from, eff_to
FROM   link
WHERE  (left IN (SELECT obid FROM rtbl)
  AND   right IN (SELECT obid FROM rtbl))
  AND  (SELECT COUNT(*) FROM rtbl WHERE type='assy')<=10
ORDER BY 1,2
```

In our example the tree contains only five assemblies, so the entire tree would be returned.

5.4. Maximal Improvement

Now we want to estimate the benefit we can achieve with the recursive SQL approach. In contrast to the navigational access method, we need only *one* query and receive only *one* result set. So two communications between the client and the database server are sufficient.

The data volume itself is reduced significantly: On the one hand, only those objects are transferred which are visible to the user. On the other hand, by reducing the number of queries the number of packets transferred for querying (each query uses at least one packet) could be minimized,

Table 4. Response times for multi-level expands with recursive queries

$size_{packet} = 4\text{kB}$ $\phi size_{node} = 512\text{Byte}$	$\tau = 3,$ $\nu = 9,$ $\sigma = 0.6$ MLE	$\tau = 9,$ $\nu = 3,$ $\sigma = 0.6$ MLE	$\tau = 7,$ $\nu = 5,$ $\sigma = 0.6$ MLE
T_{Lat} = 0.15	0.30	0.30	0.30
dtr = 256	3.19	7.13	51.42
$T_s = \Sigma$	3.49	7.43	51.72
saving in %	96.48	96.75	96.93
T_{Lat} = 0.15	0.30	0.30	0.30
dtr = 512	1.59	3.56	25.71
$T_s = \Sigma$	1.89	3.86	26.01
saving in %	97.59	97.87	98.05
T_{Lat} = 0.05	0.10	0.10	0.10
dtr = 1024	0.80	1.78	12.86
$T_s = \Sigma$	0.90	1.88	12.96
saving in %	96.97	97.24	97.42

too. As the recursive query may become quite large, we have to bear in mind that the query potentially needs more than one packet to be transmitted to the server. Therefore q_r in formula (5) denotes the number of packets needed to transmit the query instead of the number of queries (as in formulas (1), (2), and (3)). So the data volume and the response time for an action retrieving an object tree are as follows:

$$vol_r = q_r * size_p + n_v(t) * \ \phi size_n + q_r * \frac{1}{2} size_p \quad (5)$$

$$T_r = 2 * T_{Lat} + vol_r / dtr \quad (6)$$

In our examples (see table 2) the action's response time shrinks down to the values shown in table 4. As this approach only addresses actions involving an object tree instead of a single object, only the column of the MLE action is shown. The benefit gained amounts to more than 95 percent in all examples! Just as desired, the latency time now only plays a minor role compared to the delay caused by the data transfer.

Conclusion: If a user action is to retrieve an entire tree from a database, the combination of recursive querying together with early rule evaluation can significantly reduce response times.

5.5. Adding Rules to Queries

In the last sections the transformation of conditions into SQL-conformal clauses has been discussed. Now we will show in more detail (1) how the application rules are introduced into the system, (2) when the transformation of conditions is performed, and (3) how the recursive queries have to be modified in order to gain benefit from early rule evaluation.

Rules are introduced into a system by authorized users only (for example administrators). Typically new rules are necessary if a new user is registered. The introduction of new object types and actions also requires the creation of new rules, but this is only necessary when extensions to the PDM system (product update or customizations) are performed.

In order to create a new rule, the administrator has to choose the user, object type, and action and to enter the desired condition which is subsequently translated into the SQL-conformal representation (cf. section 5.3). Translated conditions are stored – together with the four components defining the rule – in an appropriate data structure (e.g. a table, called "rule table" in the following) at each client. In the following we will assume that a flag qualifies the different condition types. In order to perform the necessary modifications of recursive queries this rule table is used as follows:

A. Handling of ∀rows conditions

1. Fetch all relevant[9] ∀rows conditions.
2. Form the disjunction of all conditions found.
3. Append that condition to the WHERE clauses (using "AND") of *all* SELECT statements *outside* the recursive part of the query.

B. Handling of tree-aggregate conditions

4. Fetch all relevant tree-aggregate conditions.
5. Form the disjunction of all conditions found.
6. Append that condition to the WHERE clauses (using "AND") of *all* SELECT statements *outside* the recursive part of the query.

C. Handling of ∃structure conditions

7. Fetch all relevant ∃structure conditions.
8. Group the conditions by object type O (cf. 5.3.2).
9. Form the disjunctions of all conditions within the same group.
10. Append disjunctions to the WHERE clauses (using "AND") of SELECT statements *inside* the recursive part of the query which refer to O in their FROM clause.

D. Handling of "ordinary" row conditions

11. Fetch all row conditions according to the current user, referring to any object type t occurring in the query, and action = "access".
12. Group the conditions by object type t.
13. Form the disjunctions of all conditions within the same group.
14. Append disjunctions to the WHERE clauses (using "AND") of SELECT statements *inside and outside* the recursive part of the query which refer to t in their FROM clause.

Remark: This "procedural" description may create the impression that the incorporation of these facilities into a PDM system is rather straightforward. However, it is more complicated than it seems to be. The combination of different kinds of conditions, e.g. of ∀rows conditions and ∃structure conditions, is not trivial: The initial translation of the ∃structure conditions has to be modified according to the context of the ∀rows conditions! As the ∃structure condition now has to be evaluated outside the recursive part of the query, the structure of the original JOIN operation changes, and type information of the homogenized result tuples has to be considered. Another problem arises if the recursive query (or a part of it) is hidden in a view. As the query structure is not visible to the query modificator, the proposed modifications cannot be performed.

[9]"relevant" in this context means that the condition refers to the user, the object type, and the action under consideration

6. Achievements

Our aim was to find a solution which helps to significantly shorten response times of structure oriented user actions in PDM systems. Our computations show that early rule evaluation in combination with recursive queries will lead to acceptable results.

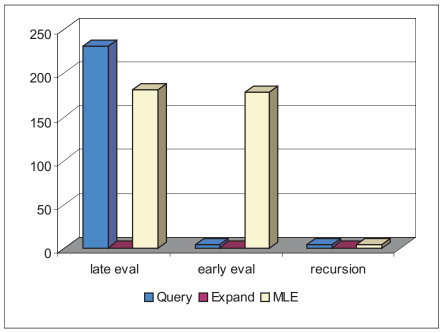

Figure 4. Response times for τ=9, ν=3, σ=0.6, T_{Lat}=150ms, *dtr*=512kBit/s

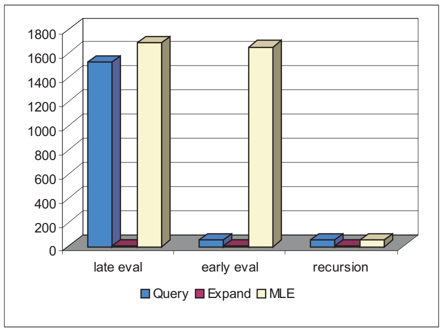

Figure 5. Response times for τ=7, ν=5, σ=0.6, T_{Lat}=150ms, *dtr*=256kBit/s

In figures 4 and 5 the results of two computation series are represented. Not very astonishing, the response time of a single-level expand does not benefit very much from our approach of early rule evaluation. Response times less than one second – without any optimizations – are already in an acceptable range. The problems are rather query actions and multi-level expands. Query actions gain significant benefit (in many cases over 95 percent) by the early evaluation of rules. The savings for the multi-level expands are very low (only two percent), however. The response times of multi-level expands only shrink to an acceptable level when combining the early evaluation of rules with recursive queries: Hereby, 95 percent and more of the original delay can be eliminated.

Unfortunately, our approach cannot solve the performance problems of all kinds of typical PDM actions. The check-out action, for example, which retrieves a structured object and prepares it for the exclusive update by one user, cannot be represented in one single query. An update of the database, setting the checked-out flag of the retrieved objects, has to be performed in a separate WAN communication. In order to avoid such additional communications, application-specific functionality performing the desired user action has to be installed at the database server (or servers if there are more than one).

In the described environment transmission costs are the dominating limitation factor. Therefore local query evaluation costs were ignored in the computational estimations. In higher bandwidth environments, however, it may be reasonable to take local query execution time into consideration.

7. Summary and Outlook

The application of PDM systems which organize all product-related data in a logically centralized manner has proved to be beneficial, especially in large companies. Therefore, company-wide usage of such systems – even in worldwide application environments – is under investigation. However, tests at DaimlerChrysler (where clients and servers were distributed between Germany and Brazil) have shown that the usage of PDM systems in such environments can lead to (extremely) long response times. As described in this paper, the cause of this problem is that the underlying relational database is used rather inefficiently: On the one hand access rules are evaluated too late, thus resulting in a large amount of unnecessarily transferred data. On the other hand user actions are translated into series of isolated SQL queries causing a large number of messages each of which burdened with the latency time of the wide-area network.

The interesting question was whether an appropriate utilization of (advanced) SQL features could help to reduce these response times to an acceptable level. That is, the focus of this investigation was to gain insights into the potentially achievable improvements. Only if these improvements are large enough, it makes sense to think about the required modifications to existing systems.

The analyses have shown that significant improvements can be achieved. In case of set-oriented queries, early rule evaluation can help to reduce the amount of transferred data (and thus response time) by orders of magnitude. The same is true for multi-level expands when utilizing early rule evaluation in combination with recursive queries facilities. Therefore it is worth to pursue this approach further and to develop strategies for the integration of these techniques into PDM systems.

This is not as trivial as it may look at first glance. For most systems this may require significant changes of major system components: It affects rule/constraint specification and evaluation, query generation, result processing, and object management. In addition, multi-server environments in conjunction with distributed data management as well as an efficient processing of check-out/check-in operations have to be taken into consideration. The treatment of these issues is beyond the scope of this paper, however, and will be subject of further investigations and prototypical implementations.

References

[1] PDM Information Center. www.pdmic.com.

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

[3] ANSI/ISO/IEC 9075-4:1999 (E). *Database Language SQL – Part 4: Persistent Stored Modules (SQL/PSM)*, September 1999.

[4] S. Banerjee and P. K. Chrysanthis. Network Latency Optimizations in Distributed Database Systems. *Fourth International Conference on DATA ENGINEERING*, pages 532–540, February 23-27, 1998.

[5] 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.

[6] J. Doppelhammer, T. Höppler, A. Kemper, and D. Kossmann. Database Performance in the Real World. *ACM SIGMOD*, 26(2):123–134, May 1997.

[7] A. Eisenberg. New Standard for Stored Procedures in SQL. *ACM SIGMOD Record*, 25(4):81–88, December 1996.

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

[9] M. J. Franklin, B. T. Jónsson, and D. Kossmann. Performance Tradeoffs for Client-Server Query Processing. In H. V. Jagadish and I. S. Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 149–160. ACM Press, 1996.

[10] IBM Corporation. *IBM DB2 Universal Database – SQL Reference – Version 6*, 1999.

[11] Informix Corporation. *Informix Guide to SQL – Tutorial*, December 1999.

[12] J. Melton. *Understanding SQL's Stored Procedures. A Complete Guide to SQL/PSM.* Morgan Kaufmann Publishers, Inc, 1998.

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

[14] A. Rosenthal, S. Heiler, and F. Manola. An Example of Knowledge-Based Query Processing in a CAD/CAM DBMS. *Proceedings of the 10th VLDB Conference*, pages 363–370, 1984.