User-Defined Table Operators: Enhancing Extensibility for ORDBMS

Michael Jaedicke
SFB342, Technische Universität München, Germany
jaedicke@in.tum.de

Bernhard Mitschang
IPVR, University of Stuttgart, Germany
Bernhard.Mitschang@informatik.uni-stuttgart.de

Abstract

Currently parallel object-relational database technology is setting the direction for the future of data management. A central enhancement of object-relational database technology is the possibility to execute arbitrary user-defined functions within SQL statements. We show the limits of this approach and propose user-defined table operators as a new concept that allows the definition and implementation of arbitrary user-defined N-ary database operators, which can be programmed using SQL or Embedded SQL (with some extensions). Our approach leads to a new dimension of extensibility that allows to push more application code into the server with full support for efficient execution and parallel processing. Furthermore it allows performance enhancements of orders of magnitude for the evaluation of many queries with complex user-defined functions as we show for two concrete examples. Finally, our implementation perception guarantees that this approach fits well into the architectures of commercial object-relational database management systems.

1. Introduction

Object-relational DBMS (ORDBMS) are the next great wave ([36], [7]). They have been proposed for all applications that need both complex queries and complex data types. Since the data volumes that come along with new data types like satellite images, videos, CAD objects, etc. are gigantic and the queries are complex, parallel database technology is essential for many of these applications. As commercial ORDBMS are based on matured parallel RDBMS, these systems are well positioned to cope with large data volumes.

But being able to handle large data volumes efficiently in parallel is not sufficient to process complex queries with short response times. For queries that apply complex algorithms to the data and especially for those that correlate data from several tables, it is essential to enable an efficient and completely parallel evaluation of these algorithms within the DBMS. However, extensions of object-relational execution systems are currently limited to user-defined functions that are invoked by built-in database operators. This concept does not provide the necessary flexibility.

Our main contribution in this paper is to propose a new approach to *user-defined database operators*. The main goals of our design were to provide extensibility with respect to new database operators, and to ensure that the design fits well to the existing technology. Especially, it should be possible to integrate the technology into current commercial ORDBMS without a major change of the system architecture. Though some ORDBMS components must be extended no component must be rewritten from scratch. Furthermore, we considered full support for parallel execution and ease of use for developers of new operators as crucial requirements.

In contrast to previous work our approach is to allow tables as arguments for user-defined routines and to allow the manipulation of these input tables by SQL DML commands in the body of these routines. Moreover, these routines are internally used as new database operators. One could at first expect that such an extension would lead to an increased complexity with respect to the development of such routines. But this is not the case, since the body of these new routines can be implemented similar to embedded SQL programs. This is a widely used programming concept.

The rest of this paper is organized as follows. We review today's concept for user-defined functions and discuss the limits of this concept in Section 2. Section 3 introduces and discusses user-defined table operators as an approach to make database systems extensible by new operators. We discuss a spatial join and an aggregation as examples of new operators in Section 4. Finally,

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

we discuss the related work in Section 5 and provide our conclusions in Section 6.

2. User-Defined Functions in ORDBMS

In this Section we provide the basic concepts and definitions that are used in this paper. We will focus on the concepts relevant to our query processing problem and refer the reader to the literature for the general concepts of relational ([12], [11]) and object-relational query processing ([36], [8]). After an introduction to user-defined functions in Section 2.1, we discuss the problem that we address here in Section 2.2.

2.1. User-Defined Functions and Predicates

Every RDBMS comes with a fixed set of built-in functions. These functions can be either scalar functions or aggregate functions. A *scalar function* can be used in SQL queries wherever an expression can be used. Typical scalar functions are arithmetic functions like + and * or `concat` for string concatenation. A scalar function is applied to the values of some columns of a row of an input table. In contrast, an *aggregate function* is applied to the values of a single column of either a group of rows or of all rows of an input table. A group of rows occurs, if a GROUP-BY clause is used. Therefore aggregate functions can be used in the projection part of SQL queries and in HAVING clauses.

In ORDBMS it is possible to use a *user-defined function* (**UDF**) at nearly all places where a system provided built-in function can appear in SQL92. Thus there are two subsets of UDFs: user-defined scalar functions (**UDSF**s) and *user-defined aggregate functions* (**UDAF**s). A UDSF that returns a boolean value and is used as a predicate in the search condition of SQL commands is a *user-defined predicate* (**UDP**). Finally, some ORDBMS [18] offer the possibility to write *user-defined table functions* (**UDTF**s), which can have (scalar) arguments of a column data type and return a table. UDTFs can be used as a table expression in SQL commands. We use the term *user-defined routines (***UDR**s*)* as a generic term for UDFs, UDTFs, and other kinds of user-defined operations like the user-defined table operators that we define later in Section 3.

2.2. Limitations of Current ORDBMS with Respect to New Database Operators

It is a well-known fact that new complex join operators can increase performance for certain operations like spatial joins [31], etc. by orders of magnitude. But, as we have already pointed out in [22], it is currently not possible for developers of database extensions to implement efficient user-defined join algorithms in current commercial ORDBMS. In fact, one cannot implement any new database operators. UDFs cannot be used to implement new operators, as they are invoked by built-in database operators. The limitation of UDTFs is obviously that - although they can produce an entire output table - they can only have scalar arguments. UDTFs are helpful in accessing external data sources [10] etc., but cannot be used to implement a new database operator like a new join algorithm. We propose a new solution for this problem in the next Section.

3. User-Defined Table Operators: UDRs with Table Arguments

When we review the existing concepts for UDRs from a more abstract point of view, we can observe the following: there are routines that operate on a tuple and produce a tuple (UDSFs), there are routines that are called for each tuple of a set of tuples and produce an aggregate value (UDAFs), and finally there are routines that operate on a tuple and return a table (UDTFs). So obviously there is something missing: namely routines that operate on one or more tables (and have possibly additional scalar parameters) and can return a tuple or a table. We want to point out that the argument tables (input tables) for this kind of routines are not restricted to be base tables. They can be intermediate results of a query as well as base tables, table expressions, or views. We call these routines *user-defined table operators* (UDTOs), since they can be used to define and implement new N-ary database operators. This classification is expressed in Table 1. As one can observe, UDTOs increase the orthogonality of SQL.

Table 1: A classification of user-defined routines based on their parameter types

		output parameter types	
		scalar	table
input parameter types	scalar	UDSF	UDTF (UDTO)
	table(s)	UDAF (UDTO)	UDTO

In the following we will explain, how UDTOs can be defined and how their processing can be integrated into parallel object-relational execution engines based on the traditional query processing framework [12]. However, we will first define a generalization relationship for row types. This will allow the application of a given UDTO to a broad range of tables, as we will see later.

3.1. A Generalization Relationship for Row Types

A row type $R = (R_1, R_2, \ldots , R_N)$ is a structured type that consists of N attributes. Each of these attributes has a data type R_i. We define that a row type $S = (S_1, S_2, \ldots , S_K)$ is a *subtype* of R, if $N \leq K$ and there is a mapping $f: \{1, \ldots , N\} \rightarrow \{1, \ldots , K\}$ such that $R_i = S_{f(i)}$ and $f(i) \neq f(j)$ for all $1 \leq i, j \leq N$. In other words, S comprises all attributes of R, but may contain additional attributes with arbitrary data types. The order of the attributes in R and S is not important. We say also that R is a *supertype* of S. Please note that each table has an associated row type.

We want to point out that this generalization relationship between subtypes is completely different from the supertable/subtable concept which describes a collection hierarchy and is

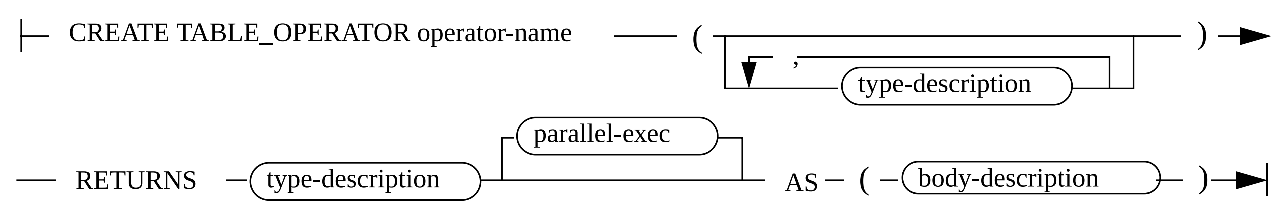

Figure 1: Syntax diagram of the CREATE TABLE_OPERATOR statement

already available in some ORDBMS [19]. As we will describe in the next Subsection, UDTOs can be defined in such a way that they are applicable to all tables whose row types are subtypes of the row types of the corresponding formal parameter input tables of the UDTO.

3.2. Defining UDTOs

3.2.1. Underlying Concept

The basic idea of this approach is easy to understand: the effect of a UDTO can be viewed as a mapping from a set of input tables to a result table or a single result tuple. This is very similar to the effect of a new algebraic operator. One fundamental difference is that a user-defined operator usually does not need to have base tables as input, but tables that represent intermediate results. It also produces an intermediate result table that can be processed further. Based on these observations, we propose to implement UDTOs by means of an extended version of embedded SQL. To enable this we propose the following extensions to user-defined routines: the definition of N input tables and a single output table is permitted and SQL DML commands in the body of this routine are allowed to refer to these input tables.

Generally speaking, a new UDTO can be sequentially executed as follows: All input tables are first materialized. That means they can be furtheron accessed in a similar way as permanently stored base tables. Then the UDTO is executed using these input tables. The UDTO produces and materializes the output table that represents the result of the UDTO and that can be processed further. Of course, the input tables cannot be manipulated and the only SQL command that is permitted for the manipulation of the output table is the INSERT command. Later, we will describe optimizations of this basic execution scheme that will allow a much more efficient execution in many cases. Moreover, in Section 3.4 we will describe how UDTOs can be processed in parallel.

We distinguish two kinds of UDTOs that differ in the implementation of their body: *procedural UDTOs* and *SQL macros.* A procedural UDTO is a UDTO whose body contains procedural statements with embedded SQL statements. As for UDSFs one can implement the body of a procedural UDTO in a programming language (with embedded SQL) compile it, put it into a library and register it with the DBMS. On the other hand, if the body of a UDTO consists of a single INSERT statement or just a RETURN statement we call this UDTO a SQL macro. This kind of UDTO has some similarity to views, but the UDTO can refer to the formal input tables of the UDTO and is therefore not limited to references to base tables or views.

3.2.2. Language Extensions

Obviously, ORDBMS must provide a statement to create UDTOs. We describe the CREATE TABLE_OPERATOR statement in the syntax diagram shown in Figure 1 (we use | to denote beginning and end of a definition; terms in small ovals are described in additional syntax diagrams or in the text). After the name of the table operator the argument list and the return type are described. The repeated use of table arguments enables the definition of N-ary table operators. The parallel execution option allows to specify how the table operator can be executed in parallel (we will refer to parallelization later in Section 3.4). Finally, the body of the routine follows. Please note, that we have not shown other options in Figure 1, which are useful for other purposes like query optimization but which are beyond the scope of this paper.

In Figure 2 we present the type description including input and output tables. Each table is described by specifying the name and data type for each column. In Figure 2 the term 'datatype' should denote all allowed data types for columns, including user-defined types. We will explain the notation `tablename.+` later.

We do not provide a syntax diagram for the description of the body, because we allow here embedded SQL program code or a single INSERT statement - with some extensions of SQL. We try to use SQL/PSM as procedural language in our examples, but our concept is not limited to a particular procedural language. That means that all procedural languages like C, C++, Java or COBOL can be used. In addition, proprietary APIs or other techniques like database programming laguages (see for example [2]) could be used instead of the traditional embedded SQL.

3.2.3. Introductory Examples

In the following we give some definitions of UDTOs. These examples are extremely simple and they are *not* intended to demonstrate the usefulness of the UDTO approach (cf. Section 4). They only serve to illustrate the concepts and the syntax. We will refer to these examples also later in Section 3.3 when we discuss the application of UDTOs.

Example 1: the UDTO `minimum`

In the first example we create a UDTO that computes the minimum for a table with an integer column:

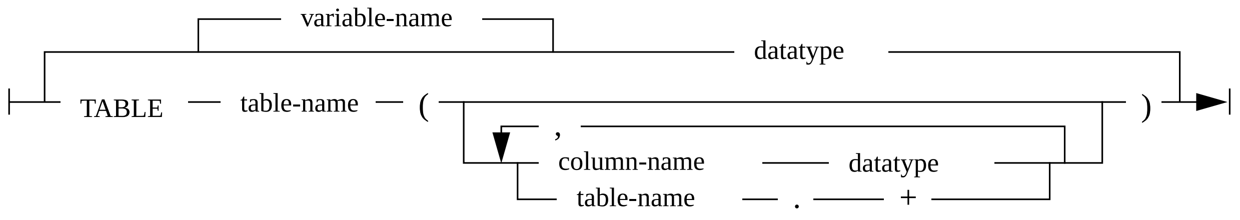

Figure 2: Syntax diagram of the type description

```
CREATE TABLE_OPERATOR minimum
(TABLE Input (number INTEGER))
RETURNS INTEGER
AS {
RETURN(SELECT MIN(value)
        FROM Input)
};
```

This example demonstrates how a new aggregation operator can be defined. Of course there are many aggregate functions (like MIN, MAX, and SUM) that should be programmed by the usual iterator paradigm, since this allows to compute multiple aggregates in a single pass over the input table. In case of aggregations, there is usually no output table, but only an aggregate value.

Before we present further examples, we first introduce the following extensions of SQL within the body of UDTOs: First, all SQL DML statements can read the input tables in the same manner as base tables. Especially, an input table can be read by several different SQL statements. Second, tuples can be appended to the output table by INSERT commands. With these extensions, we can define our next example.

Example 2: the UDTO `has_job`

This UDTO performs a restriction of the input table and does some decoding. Let us assume that a table `employees (emp_no, job)` has been defined with an integer column `job` that is used to code the job of the employees. We assume that the names of the jobs and their codes are stored in a table `jobcodes (code, jobname)`. The UDTO `has_job` selects the name for a given code from the table `jobcodes` and selects then all jobs from the input table with this code. This UDTO is created as follows:

```
CREATE TABLE_OPERATOR has_job
(TABLE Input (job INTEGER), jname VARCHAR)
RETURNS TABLE Output (job INTEGER)
AS {
INSERT INTO Output
SELECT I.job
FROM Input AS I, jobcodes AS C
WHERE  I.job = C.code AND
       C.jobname = jname
};
```

Please note that the database can be fully accessed from within the body of the UDTO. In our example the table `jobcodes` is accessed. This supports information hiding, since the accessed objects are not visible to the user of the UDTO. All side effects of a UDTO evaluation belong to the associated transaction. That is the UDTO is executed within the same transaction as the statement that invokes it.

So far UDTOs can be applied reasonably only to tables that match the row types of the corresponding formal parameter tables *exactly*. For example, the UDTO `has_job` can be applied to a table with a single INTEGER column. Of course, it is desirable to allow the application of a UDTO to a more general class of tables. Our goal is to allow all tables as input tables whose row types are subtypes of the row types of the formal input table of the UDTO. The UDTO operates then only on attributes that appear within the formal input table. All additional columns which may be present in the actual input tables are neglected or can be propagated to the output table, if this is desired (*attribute propagation*).

Therefore developers of UDTOs must have the possibility to determine that the additional columns of an actual input tuple have to be appended to the output tuple. We denote these additional columns by the expression `table_name.+` (the '+' denotes only the *additional* columns. By contrast, *all* columns are usually denoted by the '*' in SQL). That means, an expression like `table_name.+` has to be replaced by all additional columns of the corresponding input table `table_name`, which are present in the actual argument table, but not in the formal argument table of the UDTO. For example, if the actual argument table that is bound to the input table `input1` has one additional column, then `input1.+` represents exactly this column. We permit also a table variable instead of a table name in combination with '+'. Normally all additional columns of the input tables will be appended to the output table. These additional columns have to appear also in the definition of the output table (that is the row type of the formal output table is then a supertype of the row type of the actual output table).

We can now redefine the UDTO `has_job` with attribute propagation as follows (changes are in bold face):

```
CREATE TABLE_OPERATOR has_job
(TABLE Input (job INTEGER), jname VARCHAR)
RETURNS
TABLE Output (job INTEGER, Input.+)
AS {
INSERT INTO Output
SELECT I.job, I.+
FROM Input AS I, jobcodes AS C
WHERE  I.job = C.code AND
       C.jobname = jname
};
```

As the example shows, we have to define the columns of each input table, but only those columns that are needed within the routine's body should be defined. The expression `I.+` appends all additional columns to the output. This allows the application of the UDTO `has_job` as a restriction operator, because a subset of the rows of the input table is returned. The specification of the output table contains the term '`Input.+`' to enable type checking.

3.2.4. Row Identification

Finally, we want to propose here an extension that allows to implement UDTOs more efficiently. Within the body of a UDTO it can be necessary to have access to a unique identifier for each row of an input table (cf. Section 4.1 for an example). To support this, we introduce the special column data type ID for the type description of table columns that are UDTO arguments. The special type ID means that the column contains a unique identifier for each row of an input table. Note that an ID can be either a primary key or an internal identifier like a row identifier or a reference type as proposed in SQL3. Such an ID can always be generated automatically by the DBMS (this can be viewed as a kind of type promotion for row types). An ID column could also be created explicitly in the body of the UDTO by the developer, but if it is defined as a column of an input table, the DBMS can use an already existing identifier as an optimization. In general, it is not useful to append a column value of type ID explicitly to the output table. In case the primary key is used internally to represent the ID, the system does this automatically, if the '+' option has been specified in the projection list of the subquery of the INSERT statement.

3.3. The Different Usages of UDTOs

In this Subsection, we will describe two ways in which UDTOs can be used: first they can be used explicitly by programmers within SQL commands. This allows to augment the functionality of SQL in arbitrary ways. Second, UDTOs can be used to augment the implementation of database operations which involve UDFs. In this case the query optimizer has the task to use the UDTO during the plan generation whenever the use of this UDTO leads to a cheaper query execution plan. We discuss these two applications now in greater detail.

3.3.1. Augmentation of SQL

The explicit usage of UDTOs in SQL statements allows to extend the functionality of SQL by arbitrary new set operations, or to say it in other words: UDTOs make object-relational query processing universal in the sense that the set of database operations becomes extensible. For example, over time a lot of special join operations have appeared: cross join, inner join, anti-join, left, right, and full outer join, union join, etc. Moreover, other operations like recursion or more application specific ones (for data mining, etc.) have been proposed. UDTOs allow developers to define a parallelizable implementation for such operators. These operators can then be invoked in SQL commands by application programmers, as we explain in the following.

A UDTO that returns a table can be used in all places within SQL commands where a table expression is allowed. Moreover, UDTOs with two input tables can be written in infix notation to allow an easy integration into the current SQL command syntax. For example, one could define a UDTO named ANTI_JOIN. Then one can write the following expression in a FROM clause: Table1 ANTI_JOIN Table2. In this case, conceptually, the UDTO is evaluated within the FROM clause. This means that conceptually the Cartesian product of the output table of the UDTO and of all other tables, views, and table expressions in the FROM clause is computed. In addition, UDTOs can also be written in infix notation between two SELECT blocks like the usual set operations (UNION, INTERSECT, EXCEPT).

To allow the application of UDTOs to base tables and views whose row type is a subtype of the row type of the formal input table, we propose the following syntax to bind columns of the actual input table to columns of the formal input table. The programmer can specify an ordered list of columns from a given table (or view) that is bound to the corresponding columns in the parameter list of the UDTO. For example the expression TO1 (T1 USING (C_1, C_2, ..., C_N)) describes that the columns named C_1, C_2, ..., C_N of table T1 are bound to the N columns of the formal input table parameter of the UDTO TO1. The keyword USING is optional and can be left out. This notation can also be used, if binary UDTOs are written in infix notation (it can be seen as a generalization of the ON clause for traditional join expressions). If input tables are given as table expressions then the columns of the resulting table are bound to the columns of the formal input table in exactly the order in which these columns appear in the SELECT clause of the table expression.

The following statements illustrate this syntax. The first query invokes the UDTO `has_job` with a base table; the second query invokes it with a table expression:

```
SELECT *
FROM has_job(employees USING (job),'manager')

SELECT * FROM has_job(
    (SELECT job, emp_no FROM employees),
    'manager' )
```

3.3.2. Augmentation of the Implementation of UDFs

In this Subsection we describe how UDTOs can be used to improve the performance for queries with UDFs. A very important usage of UDTOs is to define more efficient database operators that can be used by the query optimizer during the plan generation. While there might be some relational queries that can be enhanced by UDTOs, the move to object-relational query processing with UDFs creates a need for UDTOs as we have already outlined in Subsection 2.2. The reason is that UDTOs allow to implement database operations that involve UDFs sometimes more efficiently than in current ORDBMS, where a built-in database operator invokes the UDF.

UDTOs provide a different implementation technique for operations involving UDFs compared to the traditional iterator-based approach for UDF evaluation. For example, a UDSF can be used as a UDP in a *join*, i.e. in a restriction involving attributes from different tables on top of a Cartesian product. In this case, a UDTO will often allow a more efficient implementation. The rea-

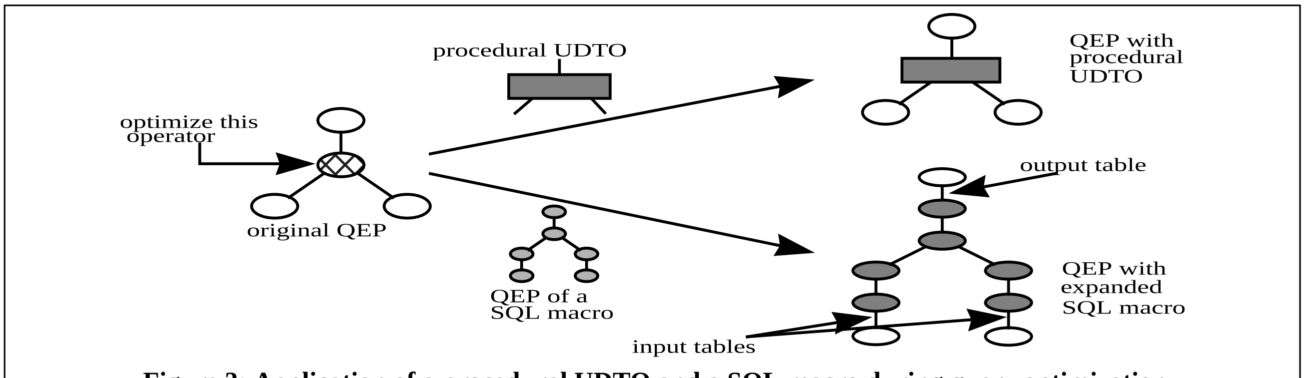

Figure 3: Application of a procedural UDTO and a SQL macro during query optimization

son is that normally the UDP will be evaluated by a nested-loops join operator which has quadratic complexity. By contrast there might be implementation methods with much lower complexity. Therefore joins are an important application of UDTOs, where performance enhancements of orders of magnitude are possible (often because nested-loops joins can be replaced by hash- or sort-merge-based join algorithms; cf. Section 4.1). Furthermore, UDTOs might sometimes be useful as aggregation, restriction, or projection operators. For example, in case of UDAFs it can be useful to provide an implementation by means of a UDTO, since this allows access to the whole input table for the aggregation (cf. Section 4.2 for an example).

The query optimizer has the task to decide when a UDTO should be used. In a rule- and cost-based query optimizer ([13], [15], [25]) this means the following: there must be a rule that generates a plan with the UDTO as an alternative implementation. Such a rule has to be specified by the developer. This is not difficult because the UDTO is always associated with a specific UDF for which it implements a specific database operator (for example a join that has exactly this UDF as join predicate). Hence, the developer must tell the query optimizer only that the UDTO can be used to evaluate the UDF. For this purpose, the CREATE FUNCTION statement that is used to register UDFs with the DBMS can be extended. The statement should include the possibility to specify that a UDTO can be used as an implementation for a specific operation such as a join. For example one could extend the CREATE FUNCTION statement as follows:

```
ALLOW <UDTO-name> AS
(JOIN | RESTRICTION | PROJECTION |
        AGGREGATION) OPERATOR
```

The relationship between the UDF and the UDTO is stored in the system tables and can be used by the query optimizer. The query optimizer has to be extended by general rules that can do these transformations for arbitrary UDFs. The optimizer uses information from the system tables to decide whether the transformation is possible for a given UDF. Please note, that for some functions like a UDAF, the UDTO might be the only implementation. In this case the UDTO is mandatory to execute the UDF.

Let us assume that we want to create a UDP `has_job` for the UDTO that we have introduced in Section 3.2. Then one can register this UDP with the UDTO `has_job` as restriction operator:

```
CREATE FUNCTION has_job (INTEGER, VARCHAR)
RETURNS BOOLEAN
ALLOW has_job AS RESTRICTION OPERATOR ...
```

After this registration the query optimizer considers the UDTO `has_job` as an implementation for the restriction with the UDP `has_job` in the following query:

```
SELECT *
FROM employees AS E
WHERE has_job(E.job, ‘manager’)
```

Figure 3 illustrates how a traditional database operator that invokes a UDF is replaced by a UDTO. First the operator has to be identified in the original query execution plan (QEP). Then the optimizer replaces this operator either by a procedural UDTO or by a SQL macro. Because the body of a SQL macro consists essentially of a QEP we can simply replace the operator by this QEP (*SQL macro expansion*). However the QEP of the SQL macro has to be modified so that it fits to the comprising QEP. For example, proper attribute propagation has to be considered. The result of this SQL macro expansion is a traditional QEP, which can be further optimized and evaluated as usual. Especially, the materialization of input and output tables can be avoided.

3.4. Parallel Processing of Procedural UDTOs

Nowadays new operators would not be a concept of great use, if these operators could not be executed in parallel. That’s why we will discuss the parallel execution of UDTOs in this Subsection. Please note that all SQL DML commands within the body of UDTOs of the implementation are parallelizable as usual. If the UDTO is a SQL-macro, i.e. a single SQL statement, the complete UDTO can be parallelized automatically by the DBMS. In the more general case of a UDTO that is implemented by embedded SQL, the developer must specify how a parallel execution can be done, if it is possible at all.

We provide a method that allows to specify, if an operator can be executed with data parallelism and, should the occasion arise, how the operator can be processed. Applying data parallelism means that one or more of the input tables of an operator are split

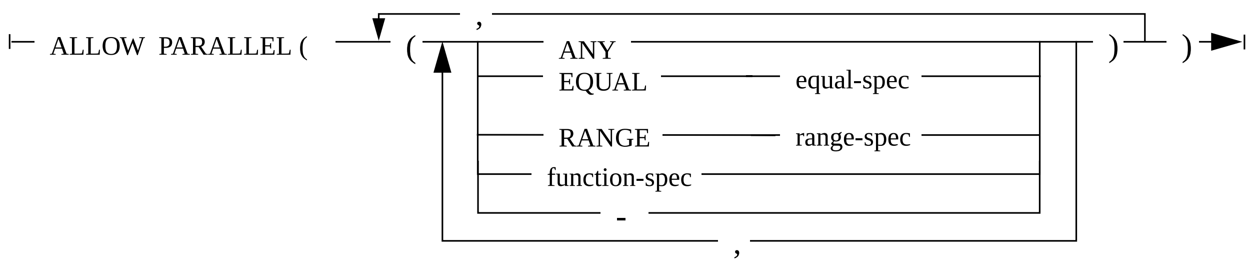

Figure 4: Syntax diagram of the parallel execution option

horizontally into several partitions by means of a suitable partitioning function. If one or more input tables can be partitioned but not all, the other input tables are replicated. Then the operator is executed once for each partition with the following input tables: if the argument table is partitionable, the respective partition of this table is used. If the argument table is not partitioned, then the complete, replicated table is used as an argument. This means that N instances of the operator are executed in parallel if one or more input tables are split into N partitions. Hence, the degree of parallelism is N. In this case, the final result is computed by combining the N intermediate output tables by means of a union (without elimination of duplicates). If no input table is partitioned, the operator is executed without data parallelism - that is sequentially. There can be several possibilities for the parallelization of an operator, depending on which input tables are partitioned and depending on how this partitioning is done.

Hence we must describe the set of all combinations of partitioning functions that can be used to partition the input tables of a given UDTO. We permit different partitioning functions for different tables, but all partitioned input tables must have the same number of partitions. Therefore the partitioning functions must have a parameter that allows to specify the number of generated partitions. This parameter enables the optimizer to set the degree of parallelism for a given UDTO. In some cases, it may be also necessary to specify that exactly the same partitioning function has to be used for some or all input tables. For example, this is needed for equi joins in relational processing.

In [21] we have already proposed the following classes of partitioning functions:

1. ANY: the class of all partitioning functions. Round-robin and random partitioning functions are examples of this class which belong to no other class. All partitioning functions that are not based on attribute values belong only to this class.
2. EQUAL (column name): the class of partitioning functions that map all rows of the input table with equal values in the selected column into the same partition. Examples of EQUAL functions are partitioning functions that use hashing.
3. RANGE (column name): the class of partitioning functions that map rows, whose values of the specified column belong to a certain range, into the same partition. Obviously there must be a total order on the data type of the column (for further details see [21]).

In addition to these three partitioning classes we have proposed that a special user-defined partitioning function can be specified, too. Based on these considerations we have developed the parallel execution option in the CREATE TABLE_OPERATOR statement that allows to specify *all* parallel execution schemes which are possible for a new operator. For operators that have multiple input tables there can be many possibilities. But since we doubt that there will be many complex operators with more than two input tables in practice, we have not tried to optimize the description for this special case. The syntax diagram for this option is shown in Figure 4.

If the partitioning class is not ANY, we have to specify the columns to which the partitioning function should be applied. The same is necessary, if a specific partitioning function must be used. We have left out these details in Figure 4. If no partitioning is possible for a given input table, this is denoted by '-' (in case of parallel processing, this input table is replicated).

In the following, we will describe some examples of parallel execution schemes for familiar (relational) operations: a restriction, a nested-loops join, a hash join, and a merge join. To simplify the examples, we have left out the column specifications of the input and the output tables and the bodies of the operators.

```
CREATE TABLE_OPERATOR restriction
(TABLE Input1(..))
RETURNS TABLE Output1(...)
ALLOW PARALLEL ((ANY))
AS { ... };
```

By specifying the class ANY in the ALLOW PARALLEL option, we have specified that all partitioning functions can be used to partition the input table for the restriction operator that is defined in this example.

```
CREATE TABLE_OPERATOR nested_loops
(TABLE Input1(...), TABLE Input2(...))
RETURNS TABLE Output1(...)
ALLOW PARALLEL ((ANY,-), (-,ANY))
AS { ... };
```

```
CREATE TABLE_OPERATOR hash_join
(TABLE Input1(...), TABLE Input2(...))
RETURNS TABLE Output1(...)
ALLOW PARALLEL
(   (EQUAL pf1(Column_List),
     EQUAL pf1(Column_List)),
    (ANY,-), (-,ANY) )
AS { ... };
```

```
CREATE TABLE_OPERATOR merge_join
(TABLE Input1(...), TABLE Input2(...))
RETURNS TABLE Output1(...)
ALLOW PARALLEL
    (  (EQUAL pf1(Column_List),
        EQUAL pf1(Column_List)),
       (ANY,-), (-,ANY) )
AS { ... };
```

The options for the three join algorithms specify that if one table is replicated, the other table can be partitioned with any partitioning function. In addition, the hash join can be parallelized with the partitioning scheme `(EQUAL pf1(Column_List), EQUAL pf1(Column_List))` that means both input tables are partitioned with the *same* partitioning function `pf1`. The same holds for the merge join, if we restrict it to the computation of equi joins. If we want to use the merge join for more general join predicates - for example an interval join (to execute a predicate like ‘x <= y + k or x >= y - k’ more efficiently) - then we need a partitioning with a function of class `RANGE(k)`, that is the option should be `ALLOW PARALLEL (RANGE(k) pf1(Column_List), RANGE(k) pf1(Column_List))`. The parameter k could be an argument of this join operator (The corresponding UDP would be used as follows: `interval_join(table1.x, table2.y, k)`).

3.5. Extension to Multiple Output Tables

We can provide even more extensibility, if we allow multiple output tables for UDTFs: one application could be to return a set of tables perhaps holding complex objects that are linked via OIDs. This is something like pushing Starburst’s XNF ([27], [33]) into the middle of SQL commands. Using such a UDTO at the top of a query would allow for composite objects as result, which might be interesting for querying XML data ([3], [4]), for example. Internally, the top operator of queries has to be extended, to allow the direct output of several tables as a result of a query. Another use of multiple output tables could be to support a nesting of complex UDTOs. The output tables of one UDTO can then serve as input tables for other UDTOs.

UDTOs with multiple output tables can be used within the FROM clause of queries but not in the WHERE clause, since they do not return a table expression. The renaming of the result tables and their columns should be allowed. UDTOs with multiple output tables can be evaluated in the same manner as UDTOs with a single output table, but they produce multiple output tables. These output tables can be processed further. The result tables in case of a parallel evaluation are obtained by a union of all corresponding partial result tables.

4. Applicability and Expressive Power of the UDTO Concept

The broad applicability and the high expressive power of the UDTO concept can be easily assessed by means of example scenarios. We present the realization of complex operations in two different processing scenarios: one is the well-known spatial join and the other refers to a complex aggregation as needed in OLAP, for example.

4.1. Computing a Spatial Join

In our first example we use the UDTO concept to define a spatial join based on the partition-based spatial merge-join (PBSM) algorithm [31]. Thus we show that the UDTO concept allows among other things a much more elegant implementation than the multi-operator method [22], which we have demonstrated using the same example scenario. We consider the following polygon intersection query (that searches for all gold deposits intersecting lakes) as a concrete example of a spatial join:

```
SELECT*
FROM Lakes as L, Gold_Deposits as G
WHEREoverlaps(L.poly_geometry,
G.poly_geometry)
```

The predicate `overlaps (polygon, polygon)` returns `TRUE`, if the two polygons overlap geometrically and `FALSE` otherwise. In order to define a UDTO for `overlaps` based on the PBSM algorithm [31] we create the following UDFs:

- `bbox(polygon)`:
  This UDSF creates and returns a bounding box (minimum bounding rectangle) for a given polygon.
- `bbox_overlaps(bbox1, bbox2)`:
  This UDP returns TRUE, if both bounding boxes overlap.
- `exact_overlaps(polygon1, polygon2)`:
  This UDP returns TRUE, if the exact geometries of the input polygons overlap.
- `bucket_no(bbox)`:
  This UDTF divides the spatial universe into B equally sized rectangular regions called buckets. Then it computes and returns all buckets, which the input bounding box intersects. Please note that this is a table function. That is it returns a table with a single column of type integer.

With these UDFs we are prepared to create the UDTO `overlaps`. This operator uses three techniques to improve the efficiency of the join algorithm. First, it uses a simple filter-and-refine scheme [30]. The filtering step uses bounding boxes as approximations of the polygons. This means that we test whether the bounding boxes overlap, before we check whether the exact geometries overlap. Second, spatial partitioning is used. This allows to join only the corresponding buckets. Third, the exact geometry is eliminated to reduce the data volumes of the input tables of the join. For the refinement the exact geometry is retrieved. This results in the implementation of the overlaps pred-

```
CREATE TABLE_OPERATOR overlaps
(TABLE Input1(id1 ID, poly1 POLYGON), TABLE Input2(id2 ID, poly2 POLYGON))
RETURNS TABLE Output1(poly1 POLYGON, Input1.+, poly2 POLYGON, Input2.+)
AS {
INSERT INTO Output1
WITH    Temp1(id, bbox, bucket) AS
        (SELECT id1, bbox(poly1), bucket FROM Input1, TABLE(bucket_no(bbox(poly1))) AS B(bucket)),
        Temp2(id, bbox, bucket) AS
        (SELECT id2, bbox(poly2), bucket FROM Input2, TABLE (bucket_no(bbox(poly2))) AS B(bucket))
SELECT  poly1, Input1.+, poly2, Input2.+
FROM    (SELECT DISTINCT Temp1.id AS id1, Temp2.id AS id2 FROM Temp1, Temp2
         WHERE Temp1.bucket = Temp2.bucket AND
         bbox_overlaps(Temp1.bbox,Temp2.bbox)) AS Temp3, Input1, Input2
WHERE   Temp3.id1 = Input1.id1 AND Temp3.id2 = Input2.id2 AND
        exact_overlaps(Input1.poly1, Input2.poly2)
};
CREATE FUNCTION overlaps (POLYGON, POLYGON) RETURNS BOOLEAN
ALLOW overlaps AS JOIN OPERATOR ...
```

Figure 5: Definition of a SQL macro as join operator for the UDP overlaps

icate by means of a new join operator that shown in Figure 5. We discuss this implementation in the following.

The subqueries in the WITH clause generate two temporary tables with the bounding boxes and the bucket numbers for spatial partitioning. Since the UDTF `bucket_no` is used in the FROM clause with a correlated tuple variable, the Cartesian product with the corresponding tuple is generated, that is we replicate the tuple for each intersecting bucket and append the bucket number (a single polygon can intersect several buckets [31]). This allows later on to join the temporary tables on the bucket number (`Temp1.bucket = Temp2.bucket` in the innermost SELECT query). Thus the function `bbox_overlaps` is only evaluated on the Cartesian product of all polygons within the same bucket. Next, duplicate candidate pairs, which might have been introduced by the spatial partitioning, are eliminated. Finally, in the outermost SELECT statement, the UDF `exact_overlaps` is processed on the exact polygon geometries that are fetched from the input tables using an equi join on the unique values of the ID columns.

We want to add some remarks on this example. The UDTO `overlaps` is a SQL macro and can be parallelized automatically. Thus there is no need to specify an option for parallel execution. In order to process the join with data parallelism the bucket number would be selected as a partitioning column due to the equi join on the bucket number. Therefore no specific user-defined partitioning function is needed for parallel processing, as the usual partitioning functions can be applied to the bucket number which is an INTEGER value. Please note, that if the join on the bucket number is done via a hash join with linear complexity, the overall complexity of the UDTO is still linear. This is *much* better than the quadratic complexity of the Cartesian product that has to be used, if no UDTO is provided.

We did measurements with a prototypical implementation of a spatial join with a similar SQL macro and observed an improvement by a factor of 238 for a table with 10 000 polygons. Moreover, we could achieve a speedup of 2.5 for the plan with the SQL macro on a four processor SMP (cf. [23] for a detailed discussion).

A final but important point concerns the function `bucket_no` that in effect does a spatial partitioning. Actually this function is too simple for practice. The reason is that it is crucial for the performance of the algorithm to find a suitable partitioning of the spatial universe. The spatial universe is the area which contains all polygons from both relations. The task is to find a suitable partitioning of the spatial universe into buckets such that each bucket contains roughly the same number of polygons (or at least the differences are not extreme). This task is very difficult, because it should ideally take the following parameters into account: the number of polygons in each input table, their spatial location, their area, the number of points per polygon and their spatial distribution. For traditional relational queries the optimizer tries to use more or less sophisticated statistics that are stored in the system tables to estimate value distributions, etc. In the same manner one could now use such meta data from (user-defined) catalog tables to compute parameters for the spatial partitioning. However, the fundamental problem with this approach would be that the input tables do not correspond to base relations and may have therefore different value distributions (as usual there might be also the problem that the statistics might not be up to date). A more sophisticated approach would be to analyze the polygons in the input tables by extracting statistics on the bounding boxes and to use these statistics to derive an appropriate spatial partitioning. UDTOs provide full support for this method (cf. [24] for the extended example). Therefore UDTOs support *run-time optimization* that takes the actual content of the input tables into account.

4.2. Computing the Median: an Aggregation Operator

In this Section we will show how a complex aggregation operation can be implemented in a clean and efficient way as a proce-

```
SELECT MIN(Age)
FROM Persons AS P
WHERE
(SELECT  Ceiling((COUNT(*)+1/2)
 FROM Persons)
<=
(SELECT COUNT(*) FROM Persons AS R
 WHERE R.Age <= P.Age)
```

Figure 6: Computing the median in SQL

dural UDTO. As a concrete example, we consider the `Median` aggregate function that computes the $\lceil (N+1)/2 \rceil$ largest element of a set with N elements. A query finding the median of a set is not very intuitively expressible in SQL92. For example, the simple query to select the median of the ages of certain persons could be formulated as shown in Figure 6. Of course one would prefer a query using a UDAF `Median` as shown in Figure 7. This query

```
SELECT Median(P.Age)
FROM Persons AS P
```

Figure 7: Computing the median with a UDTO

is not only easy to write, but will also run more efficiently (orders of magnitude for a large input table), because the `Median` function can be implemented with lower complexity than the SQL statement in Figure 6. For example, in our UDTO the median is simply computed by fetching values from the sorted input table, until the position of the median is reached. This position is first determined by counting the input table. Here are the statements to create the UDTO `median` and the corresponding UDAF:

```
CREATE TABLE_OPERATOR median
(TABLE Input1(value INTEGER))
RETURNS INTEGER
AS {
DECLARE count, cardinality,
              median_pos, result INTEGER;
SET count = 1;
SELECT COUNT(*) FROM Input1 INTO cardinality;
SET median_pos = ceiling ((cardinality + 1) / 2);
F1:  FOR result AS
        SELECT * FROM Input1 ORDER BY value ASC
     DO
     IF (count = median_pos) THEN
        LEAVE F1;
     SET count = count + 1;
END FOR;
RETURN result;
};
```

```
CREATE AGGREGATE Median (INTEGER)
RETURNS INTEGER
ALLOW median AS AGGREGATION OPERATOR ...
```

This example demonstrates how SQL DML statements and procedural statements can be mixed in the body of a UDTO. While this implementation does not use the most efficient algorithm known to compute the median, the algorithm is easy to implement based on SQL and allows a computation for arbitrary large data sets, as it does not rely on explicit intermediate data storage in main memory. Moreover, both embedded SQL queries can be evaluated in parallel as usual. That means that the optimizer can automatically decide to perform the sort operation for the ORDER BY clause in parallel. This example shows again, how our technique can enable a parallel execution of complex user-defined operators. This is a significant progress compared to other approaches. Implementing the median as an aggregate function based on the usual iterator paradigm for UDAFs is much more difficult as we have already pointed out in [21]. Using a first prototypical implementation of procedural UDTOs in an ORDBMS, we computed the median on a table with 20 000 tuples and measured an improvement by a factor of 2550 by means of a UDTO (cf. [23] for details).

5. Related work

User-Defined Functions (UDFs) have attracted increasing interest of researchers as well as the industry in recent years (see e.g. [1], [9], [16], [17], [26], [28], [29], [34], [35], [36]). However, most of the work discusses only the non-parallel execution of UDFs, special implementation techniques like caching, or query optimization for UDFs. In [32] support for the parallel implementation of UDFs in the area of geo-spatial applications is discussed. It is remarked that in this area complex operations are commonplace. Also special new join techniques [31] and other special implementation techniques have been proposed, but no framework for extensibility that allows the integration of such special processing in parallel ORDBMS was mentioned.

An approach that offered extensibility by means of new database operators and that is superior in functionality to our approach is that of the EXODUS project [6]. In EXODUS new operators could be programmed with the E programming language. However, the EXODUS approach differs from our approach fundamentally, since the goal of EXODUS was not to provide a complete DBMS. Rather the goal was to enable the semi-automatic construction of an application specific DBMS. Thus EXODUS was a database software engineering project providing software tools for DBMS construction by vendors. By contrast our approach allows to extend a complete ORDBMS by third parties like independent software vendors. We believe that our approach to program new operators with embedded SQL statements provides more support for parallel execution and fits well into current system architectures. In addition, developers can use a familiar technique to program UDTOs. UDTOs are less flexible than built-in database operators, because they cannot be applied to tables with arbitrary row types. However, they fit perfectly to UDFs. Hence they are the ideal concept to support database extensions for class libraries by third parties as well as application specific extensions. See [14] for a formal approach to the specification of database operations.

In [34] E-ADTs are proposed as a new approach to the software architecture of ORDBMS. An ORDBMS is envisioned as a collection of E-ADTs (enhanced ADTs). These E-ADTs encapsulate the complete functionality and implementation of ADTs. We believe that this is an interesting approach that is in general more ambitious than UDTOs. In contrast to the E-ADT approach, UDTOs fit very well into the architectures of current commercial ORDBMS. Thus UDTOs leverage existing technology. Moreover, UDTOs are designed to support parallel execution.

We have already mentioned that SQL macros can be viewed as a generalization of views [37]. The difference is that views can refer only to existing base tables and other views, but not to the results of subqueries or table expressions and that views cannot have parameters. As we have described, SQL macros can be used to implement database operations with UDFs more efficiently. Hence, SQL macros differ in their functionality from views.

In [21] we proposed a framework for parallel processing of user-defined scalar and aggregate functions in ORDBMS. We introduced the concept of partitioning classes there to support the parallel execution of user-defined scalar and aggregate functions. In this paper we have generalized this work to enable data parallelism for N-ary user-defined table operators. In [22] we proposed the multi-operator method to allow the implementation of complex UDFs like parallel join algorithms for UDPs. However, we view UDTOs in the form of SQL macros as the more appropriate implementation technique. Moreover, procedural UDTOs are a much more powerful concept than the multi-operator method.

6. Summary, Conclusions and Future Work

In this paper we have proposed UDTOs as a novel approach to extensibility with regard to the execution engine and the query optimizer of ORDBMS. While current user-defined functions are used within the traditional database operators, our approach allows to develop user-defined database operators.This technology will provide a new dimension of extensibility for ORDBMS.

We have presented the following core issues of UDTOs:

- the possibility to define M input tables and N output tables for a user-defined routine
- the access to and the manipulation of these tables by means of SQL commands that are embedded into procedural code (procedural UDTOs) or by means of a single SQL statement (SQL macro)
- attribute propagation to allow the application of UDTOs to a broad range of input tables based on a generalization relationship between row types
- a method to specify parallel execution schemes for UDTOs and the general algorithm for their parallel processing
- the explicit application of UDTOs within SQL and their use as high performance implementations for operations involving UDFs.

We believe that the possibility to define new operators is very promising, especially since the SQL-based implementation technique is in our view elegant and easy to understand for developers. In addition, sophisticated optimization technology can be used to produce high-quality plans that are automatically fine tuned to the estimated data volumes.

With regard to SQL macros the UDTO approach is similar to pushing views into the middle of SQL statements. SQL macros allow to push code into a new operator, where it is defined once (e.g. in a DBMS class library) and available for general use in SQL. Hence only a single definition has to be maintained. This eases the task of the application programmer, makes it less error-prone, improves the declarative character of SQL DML commands and enhances the readability. Moreover, SQL macros can always be completely integrated into the query execution plans of SQL statements by macro expansion. As a consequence, the usual parallelization techniques can be used.

The concept of procedural UDTOs is much more powerful, because one can execute a query on the input tables and use a procedural language like SQL PSM to implement complex code. This is especially of interest in combination with an API that is provided for the development of DBMS class libraries by some ORDBMS ([19], [20]). This offers the possibility to implement new algorithms like join algorithms, for example. Moreover, our approach supports data parallelism for these new database operators. Besides being able to define parallel processing schemes by specifying allowed partitioning functions, the possibility to use SQL goes a long way towards enabling as much parallelism as possible, since all embedded SQL statements can be processed automatically in parallel. An additional advantage of our SQL-based approach to the implementation of UDTOs is that query optimization can be fully exploited.

Areas of future work are optimization issues for UDTOs and case studies for their application in other scenarios: promising areas of interest are OLAP, data mining, image analysis, time series processing, genome analysis, or querying XML, for example. Moreover, if several SQL statements are used in the body of the UDTO, multi-query optimization techniques could be beneficial. First results can be found in [24].

Currently, an implementation of UDTOs in MIDAS [5], a prototype of a parallel ORDBMS, is under way. The core extensions have been completed and in a future paper [23] we will report on this effort and describe implementation concepts for UDTOs. As we have mentioned, first measurements demonstrated performance improvements of orders of magnitude by means of UDTOs.

Acknowledgments

We gratefully acknowledge the help of our master student Sebastian Heupel with the implementation and his comments on the draft of this paper which helped us to improve the presentation. We also acknowledge the cooperation with the complete MIDAS team and the valuable comments from the referees.

7. References

[1] Antoshenkov, G., Ziauddin, G.: Query Processing and Optimization in Oracle Rdb. VLDB Journal 5(4): 229-237 (1996).

[2] Bancilhon, F., Buneman, P. (Eds.): Advances in Database

Programming Languages. ACM Press / Addison-Wesley 1990, ISBN 0-201-50257-7, Papers from DBPL-1, September 1987, Roscoff, France.

[3] Beech, D.: Position Paper on Query Languages for the Web, Oracle Corp., http://www.xml.com/xml/pub/Guide/Query_Languages.

[4] Bosworth, A. et al.: Microsoft's Query Language 98 Position Paper, Microsoft Corp., http://www.xml.com/xml/pub/Guide/Query_Languages.

[5] Bozas, G., Jaedicke, M., Listl, A., Mitschang, B., Reiser, A., Zimmermann, S.: On Transforming a Sequential SQL-DBMS into a Parallel One: First Results and Experiences of the MIDAS-Project, Proc. of 2nd Int. Euro-Par Conf., LNCS 1123, Springer, 1996.

[6] Carey, M. J., DeWitt, D.J., Graefe, G., Haight, D. M., Richardson, J. E., Schuh, D. T., Shekita, E. J., Vandenberg, S. L.: The EXODUS Extensible DBMS Project: An Overview, in: Zdonik, S., Maier, D. (eds.): Readings in Object-Oriented Databases, Morgan-Kaufmann, 1990.

[7] Carey, M. J., Mattos, N., Nori, A.: Object-Relational Database Systems: Principles, Products, and Challenges (Tutorial). SIGMOD 1997: 502.

[8] Chamberlin, D.: A Complete Guide to DB2 Universal Database, Morgan Kaufman Publishers, San Francisco, 1998.

[9] Chaudhuri, S., Shim, K.: Optimization of Queries with User-defined Predicates. VLDB 1996: 87-98.

[10] Deßloch, S., Mattos, N.: Integrating SQL Databases with Content-Specific Search Engines. VLDB 1997: 528-537.

[11] DeWitt, D., Gray, J.: Parallel Database Systems: The Future of High Performance Database Systems, In: CACM, Vol.35, No.6, 85-98, 1992.

[12] Graefe, G.: Query Evaluation Techniques for Large Databases. Computing Surveys 25(2): 73-170 (1993).

[13] Graefe, G.: The Cascades Framework for Query Optimization. Data Engineering Bulletin 18(3): 19-29 (1995).

[14] Güting, R. H.: Second-Order Signature: A Tool for Specifying Data Models, Query Processing, and Optimization, SIGMOD Conference 1993: 277-286.

[15] Haas, L. M., Freytag, J.C., Lohman, G. M. , Pirahesh, H.: Extensible Query Processing in Starburst. SIGMOD 1989: 377-388.

[16] Hellerstein, J. M., Stonebraker, M.: Predicate Migration: Optimizing Queries with Expensive Predicates. SIGMOD 1993: 267-276.

[17] Hellerstein, J. M., Naughton, J. F.: Query Execution Techniques for Caching Expensive Methods. SIGMOD 1996: 423-434.

[18] IBM DB2 Universal Database SQL Reference Version 5, Document Number S10J-8165-00, 1997: 441-453.

[19] Illustra User's Guide, Illustra Information Technologies, Inc., 1995.

[20] Informix Universal Server, DataBlade API Programmer's Manual Vers. 9.12, Informix Software Inc., 1997.

[21] Jaedicke, M., Mitschang, B.: On Parallel Processing of Aggregate and Scalar Functions in Object-Relational DBMS, SIGMOD 1998: 379-389.

[22] Jaedicke, M., Mitschang, B.: The Multi-Operator Method for the Efficient Parallel Evaluation of Complex User-Defined Predicates, Technical Report, to appear 1999.

[23] Jaedicke, M., Zimmermann, S., Nippl, C., Mitschang, B.: The Implementation of User-Defined Table Operators in MIDAS, (submitted) 1999.

[24] Jaedicke, M.: New Concepts for Parallel Object-Relational Query Processing, Ph.D. Thesis, University of Stuttgart, 1999.

[25] Lohman, G. M.: Grammar-like Functional Rules for Representing Query Optimization Alternatives. SIGMOD 1988: 18-27.

[26] Mattos, N., Deßloch, S., DeMichiel, L., Carey, M.: Object-Relational DB2, IBM White Paper, July 1996.

[27] Mitschang, B., Pirahesh, H., Pistor, P., Lindsay, B. G., Südkamp, N.: SQL/XNF - Processing Composite Objects as Abstractions over Relational Data. ICDE 1993: 272-282

[28] O'Connell, W., Ieong, I.T., Schrader, D., Watson, C., Au, G., Biliris, A., Choo, S., Colin, P., Linderman, G., Panagos, E., Wang, J., Walters, T.: Prospector: A Content-Based Multimedia Server for Massively Parallel Architectures. SIGMOD 1996: 68-78.

[29] Olson, M. A., Hong, W. M., Ubell, M., Stonebraker, M.: Query Processing in a Parallel Object-Relational Database System, Data Engineering Bulletin, 12/1996.

[30] Orenstein, J. A.: A Comparison of Spatial Query Processing Techniques for Native and Parameter Spaces. SIGMOD Conf. 1990: 343-352.

[31] Patel, J. M., DeWitt, D. J.: Partition Based Spatial-Merge Join. SIGMOD Conf. 1996: 259-270.

[32] Patel, J., Yu, J. Kabra, N., Tufte, K., Nag, B., Burger, J., Hall, N., Ramasamy, K., Lueder, R., Ellman, C., Kupsch, J., Guo, S., DeWitt, D. J., Naughton, J.: Building A Scalable GeoSpatial Database System: Technology, Implementation, and Evaluation, SIGMOD 1997: 336-347.

[33] Pirahesh, H., Mitschang, B. Südkamp, N., Lindsay, B. G.: Composite-Object Views in Relational DBMS: An Implementation Perspective. EDBT 1994: 23-30.

[34] Seshadri, P., Livny, M., Ramakrishnan, R.: The Case for Enhanced Abstract Data Types. VLDB 1997: 66-75.

[35] Stonebraker, M.: Inclusion of New Types in Relational Data Base Systems. ICDE 1986: 262-269.

[36] Stonebraker, M., Brown, P., Moore, D.: Object-Relational DBMSs, Second Edition, Morgan Kaufmann Publishers, 1998.

[37] Stonebraker, M.: Implementation of Integrity Constraints and Views by Query Modification. SIGMOD Conf. 1975: 65-78.