TECHNISCHE
UNIVERSITÄT
MÜNCHEN

INSTITUT FÜR INFORMATIK

Sonderforschungsbereich 342:
Methoden und Werkzeuge für die Nutzung
paralleler Rechnerarchitekturen

A Framework for Parallel Processing of Aggregate and Scalar Functions in Object-Relational DBMS

Michael Jaedicke, Bernhard Mitschang

TUM-I9741
SFB-Bericht Nr. 342/25/97 A
September 97

TUM–INFO–09-I9741-350/1.–FI

Anforderungen an: Prof. Dr. A. Bode
Sprecher SFB 342
Institut für Informatik
Technische Universität München
D-80290 München, Germany

Druck: Fakultät für Informatik der
Technischen Universität München

A Framework for Parallel Processing of Aggregate and Scalar Functions in Object-Relational DBMS[1]

Michael Jaedicke, Bernhard Mitschang
Department of Computer Science
Technische Universität München
D - 80290 Munich, Germany
e-mail: jaedicke@informatik.tu-muenchen.de

Abstract

Nowadays parallel object-relational DBMS are envisioned as the next great wave but there is still a lack of efficient implementation concepts for some parts of the proposed functionality. One of the current goals for parallel object-relational DBMS is to move towards higher performance. In our view the main potential for performance increases lies in providing additional optimization and execution information for ADTs, as queries can be executed much more efficiently if the developer teaches the DBMS part of the ADT semantics. Based on this insight we develop a framework that allows to process user-defined functions using data-parallelism, a topic not covered up to now. We describe the class of partitionable functions that can be processed in parallel with a good speedup. We also propose an extension which allows to speedup the processing of another large class of functions using parallel sorting. Functions that can be processed using our framework are for example often used in decision support queries on large data volumes like e.g. data warehouses. Hence a parallel execution is indispensable.

1. Introduction

According to [15] object-relational DBMS (ORDBMS) are the next great wave in Database Management Systems. ORDBMS are proposed for all applications that need both complex queries and complex data types. Typical ORDBMS application areas are e.g. multi-media and image applications [33], geographic information systems [14], and management of time series [27] and documents [29]. Many of these applications pose high requirements with respect to functionality and performance on ORDBMS. Thus ORDBMS need to exploit parallel database technology. These observations have recently led to significant development efforts for parallel ORDBMS (PORDBMS) of some database vendors ([7], [13], [12], [28]). Although first industrial implementations enter the marketplace and the SQL3 standard [10] is maturing, there are still many topics left for research in this area ([8], [9], [5], [35]).

One of the current goals for PORDBMS is to move towards a framework for constructing parallel ADTs[35] and more sophisticated query optimization and execution ([35], [24]). ADT functions are completely opaque for the query optimizer and thus allow only very restricted query optimization and execution techniques if no further optimization and execution information is provided. Additional information allows more sophisticated query optimization and execution as the ORDBMS knows and understands at least part of the semantics of the ADT. This will allow great performance improvements ([15],

1.This research was partially supported by the DFG, SFB 342/B2.

[24]). While there are different approaches to reach this goal ([23], [6], [24]), most ORDBMS vendors currently offer developers some parameters to describe the semantics of user-defined functions. Thus in a more abstract view the developer has the task to specify to which class a new function belongs. The system can then apply optimization and execution techniques that are appropriate for the respective class of functions. This allows an optimization and execution of queries with ADT functions using a classification. Note that parameters that are used to classify functions can be related to their semantics as well as to their specific implementation. We view this development of a *classification based query processing* as an important step towards truely sophisticated query processing in ORDBMS.

Our main contribution in this paper is to enable data parallelism for a broad class of user-defined functions. Especially we show how user-defined aggregate functions - often called set or column functions - can be computed in parallel. To this aim we propose a framework covering both the necessary interfaces that allow the appropriate registration of user-defined aggregate functions with the ORDBMS and their parallel processing. Parallel computation of user-defined aggregate functions is especially useful for application domains like decision support (e.g. based on a data warehouse that stores traditional as well as non-traditional data, like spatial, text or image data), as decision support queries often must compute complex aggregates. For example, it has been noted that in the TPC-D Benchmark 15 out of the 17 queries contain aggregate operations [17]. In addition, if scalar functions with a global context are processed in parallel, caution is needed in order to get semantically correct results. Our framework can help in this case, too. We further show that many non-partitionable functions can be easily implemented if their input is sorted and thus profit from parallel sorting.

In section 2 we give the necessary background on user-defined functions as well as some related parallel and object-relational query processing techniques. Our framework for parallel processing of user-defined functions is introduced in section 3. We also give some examples of typical functions to validate our concepts. In section 4 we discuss implementation and performance issues related to our framework. After a discussion of related work in section 5, the closing section 6 contains a short summary and a brief outlook to future work. An appendix contains a brief example of a practical application of our concepts to a real-life document management system.

2. SQL Aggregate Functions and Parallel Object-Relational Query Processing

We will now provide the basic concepts and definitions we use in this paper. We will concentrate on the concepts for our specific query processing problem and refer the reader to the literature for the general concepts of parallel relational database query processing ([1], [2], [3], [26]) and object-relational query processing ([15], [6]).

2.1. Built-in Functions and User-Defined Functions

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. Functions for type casting are special scalar functions, too. A scalar function is applied to the values of some columns of the 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. Thus aggregate functions can be used in the projection part of SQL queries and in HAVING clauses. If the optional keyword DISTINCT is specified for the argument of an aggregate function in a query, duplicate values are eliminated before the values are passed to the aggregate function. This means that the keyword DISTINCT is not considered an argument of the function, but rather a specification of an operation that is performed before the function is applied.

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 (UDSFs) and user-defined aggregate functions (UDAFs).

2.2. User-Defined Scalar Functions

Let us now describe briefly how UDSFs are created in ORDBMS. Users can write UDSFs as so-called *external* functions in a 3GL (typically C is supported as a language) and then register them with the DBMS. Figure 1 provides an example of the syntax used in DB2 for common servers [6] to register a UDF `distance` that computes the euclidean distance between two arguments of the user-defined data type `point.`

```
CREATE FUNCTION distance (point, point)
RETURNS double
EXTERNAL NAME ‘point!distance’
LANGUAGE C
PARAMETER STYLE DB2SQL
NOT VARIANT
NOT FENCED
NOT NULL CALL
NO SQL
NO EXTERNAL ACTION
NO SCRATCHPAD
NO FINAL CALL;
```

Figure 1: Registration of a new external UDSF `distance` in DB2

As can be seen from this example there are already some parameters allowing the user to describe the characteristics of a newly registered function. We refer the reader to [6] for most of the details and provide only the information of relevance to our problem. An interesting feature is the possibility to use a so-called scratchpad area for UDFs. A scratchpad area is a small piece of memory that is passed to a UDF with all calls and that is not deleted after the function execution returns the control. Thus it is possible for a function to maintain a global context (or global state), i.e. information can be preserved from one function invocation to the next. After the last call to the function within an SQL query the scratchpad is deallocated by the system. Please note that the user can allocate more memory than the rather small scratchpad area by simply allocating some memory dynamically and hooking it up in the scratchpad.

After a function has been registered, the developer of a UDF should provide the query optimizer with some information about the expected execution costs of a UDF. ORDBMS have to provide a suitable interface for this purpose. For example DB2 allows to specify the I/O and CPU costs that are expected for the first call to a function, for each further call, and per argument byte read. In addition the percentage of the UDF argument’s storage size that is processed at the average call should be specified. If a UDF is used as a predicate (i.e. the function returns a boolean value) the user should be able to specify a user-defined selectivity function [15]. As providing these details can be a time consuming task, easy to use development kits may be offered for this task [7] [28] [30].

In advanced object-relational systems it should be possible to implement the body of UDFs using SQL statements embedded in the code of a 3 GL (similar to the usual embedded SQL offered for application development). This allows access to the database in the function's body. One restriction is that a function should not modify the database if it is used in a SELECT statement. A UDF can also have an external effect that is it might perform an external action, e.g. read from or write to a file, send an email to the DB administrator, start a program, etc. The Informix Illustra DBMS supports already UDFs that consist of one or more SQL statements.

2.3. User-Defined Aggregate Functions

The aggregate functions of the SQL92 standard are `MAX`, `MIN`, `AVG`, `SUM` and `COUNT`. Other statistical aggregate functions like `standard deviation` and `variance` are provided by some RDBMS implementations [6]. Sometimes one can find also physical functions like `center_of_mass` and `angular_momentum` and functions like `volatility`, `Alpha`, `Beta`, etc. for financial analysis [21].

First of all let us look at some examples that show the need for UDAFs. If a user defines new abstract data types in an ORDBMS typically some new aggregate functions will also be needed. For example a user may define an ADT `money` together with some subtypes (e.g. `US_Dollar`, `Can_Dollar`, `S_Frank`, `J_Yen`, etc.). Now a new UDAF is needed e.g. to compute the sum of amounts in `US_Dollar`. Thus the user has to define the following new aggregate function (shown in an Informix-like syntax):

```
CREATE AGGREGATE Sum(US_Dollar) RETURNS US_Dollar ...
```

This function actually overloads the built-in aggregate function `SUM`. In the same way one could e.g. define an aggregate function `Shape_sum` that computes the geometric union of polygons. Last but not least a user may want to define new aggregate functions for existing data types like `INTEGER`.

```
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 2: Computing the median in relational SQL

For a concrete example consider the `Median` function that computes the $\lceil (N+1)/2 \rceil$ largest element of a set with N elements (that element could be informally called the "halfway" element). 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 2. Of course one would prefer the following query using the UDAF `Median` as shown in Figure 3. This query is not only easy to write but will also run more efficiently, because the `Median` function can be implemented with lower complexity than the SQL statement in Figure 2 (see section 3.1 for details).

Now it should be quite clear that UDAFs are both needed and useful in ORDBMS. Let us now see how, for example, the Informix Illustra ORDBMS supports UDAFs. The system computes aggregate functions

```
SELECT Median(P.Age, COUNT(*))
FROM Persons AS P
```

Figure 3: Computing the median in object-relational SQL

in a tuple-at-a-time fashion, i.e. there is one function call for each element of the input set. The user has to write the three following functions to implement a new UDAF:

1. *Init*(&handle):
   The Init function is once called without parameters before the actual computation of the aggregate begins. It initializes the aggregate computation and allocates the handle.

2. *Iter*(&handle, value):
   The Iter function is called once for each element of the input set. It aggregates the next value into the current aggregate that is stored using the handle.

3. value = *Final*(&handle):
   The Final function is called once after the last element of the input set has been processed by the Iter function. It computes and returns the resulting aggregate saved in the handle and deallocates the handle as well.

The handle, similar to the scratchpad area in [6], allows to store the global context of the computation, e.g. to compute the average of a set of values the *Iter* function would store both the sum of all values seen so far and their number as intermediate results in the handle. The *Final* function divides the sum by the number and returns the result. As can be seen from the description of the *Init* and *Final* functions there is one handle for a single aggregate function in a query. Obviously this design matches the usual Open-Next-Close protocol [16] for relational operators very well. After the three functions have been registered with the ORDBMS (cf. Figure 1), the user can create the aggregate function (e.g. average) using a `CREATE AGGREGATE` statement. This statement determines, which three functions are used to implement the *Init*, *Iter* and *Final* functions for the new aggregate function.

2.4. Parallel Processing of User-Defined Functions and Global Contexts

In many cases when a query is evaluated, UDSFs can be executed in parallel using data parallelism. Instead of executing a set of function invocations in a sequential order one simply partitions the data set and performs the UDF invocations for each such data partition in parallel. Obviously aggregate functions cannot use this approach without modification as they deliver only one result for a set of input tuples. In addition it is not possible to process all UDSFs in parallel using simply data parallelism. The problem is that some scalar functions have a global context. Examples are functions that use a scratchpad, functions that use SQL DML as a side effect and functions that have an external action (e.g. functions that access a file). In all these cases the semantics of the parallel execution is only correct if the result remains always the same as for sequential execution. If data parallelism is applied in the usual way then one global context is introduced for each data partition that is processed in parallel by the DBMS. In most cases the result will be semantically incorrect. Thus special support is needed to process some UDSFs as well as UDAFs in parallel.

We can identify two kinds of global contexts that have to be treated differently in query processing for UDFs. We call the first kind of global contexts *input-related*. In this case the global context is based on the input table on which the function operates. The second kind of contexts are global *external* contexts. This kind of context is introduced when DML statements or external actions are allowed in UDFs. In this case the context of the UDF might depend on arbitrary data in the database or elsewhere and the context itself may be stored in the database or outside of the control of the DBMS. Of course a function can have a context that is both, input-related and external. In this paper we will be mainly concerned with functions that have only an input-related context, if not stated otherwise. The reason is that we do see only limited opportunities for database technology to enable parallelism for functions that have a global external context. Essentially it is the responsibility of the developer to specify whether parallel execution of a function with an external context is semantically correct or not. The mechanisms to support parallelism for functions that have an input-related context can be applied orthogonally as will be detailed later in this paper.

Now we have shown that UDFs and especially UDAFs can often not be processed in parallel without special support by the DBMS. This will result in a performance bottleneck in ORDBMS query processing. In shared-nothing and shared-disk parallel architectures the input data is often distributed over various nodes and must be shipped to a single node to process the UDF correctly, i.e. sequentially, and afterwards the data possibly has to be redistributed for further parallel processing. This results in additional communication costs and therefore even worse performance.

2.5. User-Defined Functions that Operate on Sorted Tables

So far we have mainly discussed the support of existing ORDBMS for UDFs. But there is another interesting aspect that should be noted w.r.t. UDFs: many functions are only reasonable to implement if their input is sorted in a certain order and other functions require a sorted input due to their semantics. Consider for example once again the `Median` function introduced above. Obviously implementing this function in C or Java is not reasonable since the function would have to store an arbitrarily large number of input values intermediately. Possibly the best way to do this would be to use SQL and store the values in a temporary table (if this feature is supported by the ORDBMS). Finally the median must be computed somehow. Clearly this approach is costly, since it uses one SQL command per input value and has still to compute the median. A much cheaper solution as well as a much easier implementation would be possible if the user could specify that the input values have to be passed in a certain order to the function. This will be detailed in section 3.1.

There are a lot more functions that can profit from sorted input sets. In the data warehousing area, e.g. Red Brick Systems [31] offer many other aggregate functions [21] in their Red Brick Intelligent SQL query language including `Rank` and `N_Tile`. `Rank` gives the rank of a value in the set of all input values. The lowest value has rank 1. `N_Tile` returns the number of the range that contains a certain value where the ranges are computed by dividing the range of the input values in N approximately equally populated ranges.

In addition, Red Brick offers scalar functions with a global context that operate on *ordered* tables: `Cumulative` computes the sum of all values seen so far in a list, `Running_Sum` and `Running_Average` compute the sum and average respectively of the most recent N values in a list with the first N-1 return values being set to NULL. Please note, that it is not possible to sort the input values somehow in the implementation of a scalar function, since a scalar function has to produce one result for each invocation. This is an important difference to aggregate functions. In the next section we will show how a large class of user-defined functions (including functions that operate on ordered tables), can be better supported w.r.t. to an easier and more efficient implementation and how parallel processing can be enabled for their evaluation. Both techniques enable database application designers to push more application code deep into the database system. This allows a more efficient evaluation as well as the exploitation of parallelism to reduce the response times. In addition, the semantic power of UDSFs applicable to SQL is improved.

3. Parallel Processing of UDFs

In this section we describe several orthogonal approaches to enhance processing of UDFs. In subsection 3.1 we describe a technique that exploits sorting to enhance the processing of UDFs. In subsection 3.2 we define the class of partitionable functions and propose our approach for the parallel processing of these functions. In subsection 3.3 we validate our approach by examining some relational and object-relational functions and show how their implementation and execution can profit from our approach.

3.1. Allowing UDFs on Sorted Tables

As we have seen in section 2 many functions like `Median`, etc. can be easily implemented when their input is sorted according to a specified order. Some scalar functions are even only defined on sorted tables. This feature can be easily introduced by using an additional parameter when the function is registered with the ORDBMS (e.g. by adding a clause like ORDER BY, see section 4 for details of the syntax we propose). Of course the user must have the possibility to specify a user-defined order by providing a specific sort function for the argument types of the UDF that often refer to user-defined data types. In most cases such functions will be needed anyway, e.g. to support sorted query results, to build indexes (like generalized B-Trees [15] or GiSTs [32]) or for sort merge joins to efficiently evaluate predicates on user-defined data types.

Let us consider the `Median` function once again, to demonstrate how sorting can ease the implementation of functions. The `Median` function is called with two parameters(cf. Fig. 3): the first parameter is a value of the already sorted input set, the second parameter is constant and gives the cardinality of the input set. When the function is called the first time it computes the median position and stores this position in the global context. In addition the function maintains a counter of the number of invocations. During each call the function checks whether the median position is reached. In this case the function stores the input value in the global context. Because the input is sorted, this value is actually the median of the input set. Finally the function returns the median. Obviously this function is easy to implement because essentially it has to scan its input for the right position only.

One interesting point to observe is that many aggregate functions which operate on a sorted input do not need to read the complete input set to compute the aggregate. Thus it might be well worth to provide the aggregate function with the option to terminate the evaluation as early as possible and return the result. In the case of the `Median` function this option would save roughly half of the calls to the function. We call this feature *early termination*. This feature is not applicable for scalar functions because scalar functions must be applied to each row of their input table.

3.2. A Framework for Parallel Processing of UDFs

In this subsection we will define the class of partitionable functions both for scalar and aggregate functions. Our intention is to support parallel evaluation using data parallelism. The main problem here is to find a small number of communication patterns that enable parallel evaluation of the UDF using data parallelism. Such communication patterns can consist of multiple steps and can be arbitrarily complex.

Some communication patterns are widely used in RDBMS. For example a relational selection or projection operation can use data parallelism by partitioning the input data and evaluating the selection or projection for each partition in parallel as shown in Figure 4a (for a selection). Using data parallelism for

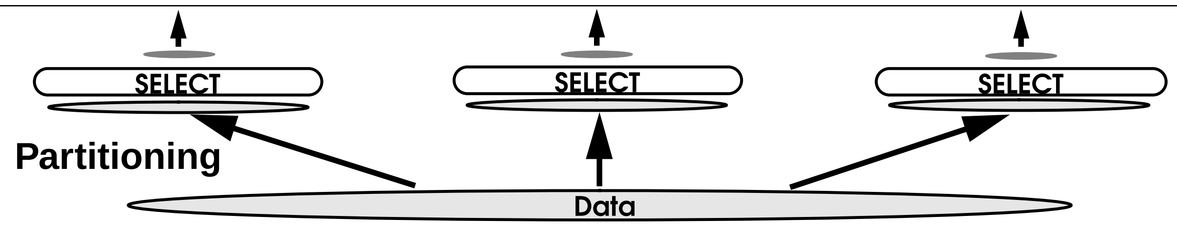

Figure 4a: Communication pattern for parallel selection in RDBMS

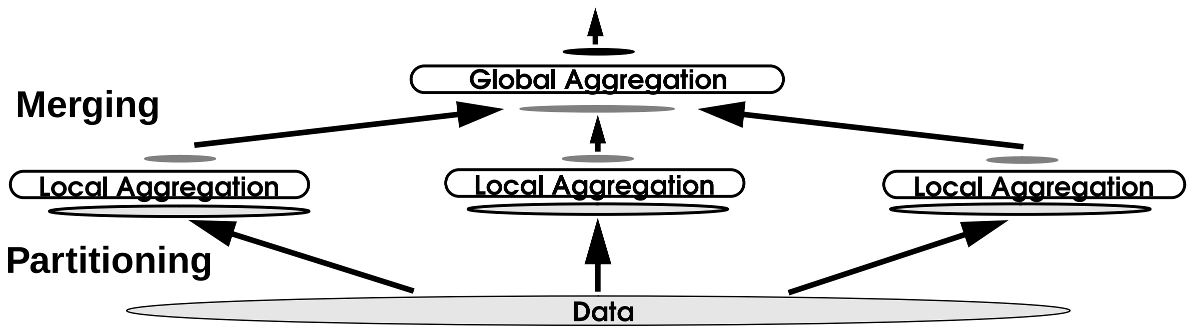

Figure 4b: Communication pattern for parallel aggregation in RDBMS

nested-loop-joins is also possible but one needs to replicate the inner (smaller) relation on all nodes that process a partition of the outer relation. Thus replication is used in combination with partitioning as a communication pattern. Parallel relational aggregation operations use a communication pattern consisting of two steps [16] as shown in Figure 4b. After the data has been partitioned it is first aggregated locally for each partition and then in a second step the locally computed sub-aggregates are combined in a global aggregation. For the `COUNT` function the local aggregation is counting while the global aggregation computes the sum of the local counts. For parallel sorting many algorithms have been proposed that sometimes need complex communication patterns between the different nodes for their implementation. The problem is that often some intermediate results (like key value distributions) have to be exchanged between all nodes.

As UDFs can have arbitrary semantics we believe that it is not possible to define a fixed set of communication patterns that allow to apply data parallelism to *all* UDFs in the most efficient way. On the other hand we currently do not see how the database system could support arbitrary user-defined communication patterns. In addition, we believe that a small number of such communication patterns is sufficient to support many functions and that these patterns have to be implemented in the ORDBMS by the database vendors. In fact, we argue in this paper that the well-known communication patterns, enhanced by some basic parametrization measures that provide the necessary extensibility, are sufficient to provide data parallelism for many user-defined functions, too.

As we already pointed out in the introduction UDFs will normally be coded in a 3GL like C and thus their semantics will be completely opaque for the SQL compiler. Thus the developer must tell the system how parallelism is applicable for a certain function, i.e. the developer has to do a classification of the function w.r.t. its ‘parallelizability’. In addition it might be necessary to implement and register additional functions to enable parallel execution, if the function belongs to certain classes.

Before we define the class of partitionable functions we want to give a taxonomy of the functions that can be used for partitioning (i.e. horizontal fragmentation). All such partitioning functions take a multi-set as input and return a partition of the input multi-set, i.e. a set of multi-sets such that any element of the input multi-set is contained exactly in one resulting multi-set. Actually we will also allow in some cases functions returning subsets that are not disjoint, i.e. functions that replicate some of the elements of the input set. We define the following increasingly more special classes of partitioning functions:

1. ANY: the class of all partitioning functions. Round-robin and random partitioning functions are examples of this class belonging 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 multi-set with equal values in the selected column into the same multi-set of the result. Examples of EQUAL functions are partitioning functions that use hashing.

3. RANGE (column name [, N]): the class of partitioning functions that map rows, whose values of the specified column belong to a certain range, to the same multi-set of the result. Obviously there must be a total order on the data type of the elements. The range of all values of the data type is split into some sub-ranges that define which elements are mapped into the same multi-set of the resulting partition. This class is especially useful for *scalar* functions that require a sorted input, and the optional parameter N is only applicable for these functions. Based on the total order of the data type the optional parameter N allows to specify that the largest N elements of the input set that are smaller than the values of a certain range have to be replicated into the result multi-set of this range. Replicated elements must be processed in a special way and are needed only to establish a “window” on a sorted list as a kind of global context for the function. Functions that can exploit this option are for example Running_Avg or Running_Sum (see section 3.3). The number of elements that belong to a certain range should be much greater than the value N.

Please note that the following inclusion property holds: RANGE $\subset$ EQUAL $\subset$ ANY. This taxonomy is useful to classify UDFs according to their processing requirements as we will see below. The database system can automatically provide at least a partitioning function of class ANY for all user-defined data types (e.g. round-robin). We define that a *class C partition* of a multi-set is a partition that is generated using a partitioning function of class C (C denotes either ANY, EQUAL or RANGE).

To simplify the presentation below, we will omit constant input parameters to UDFs in the following. Given a set $S = \{s_i \mid 1 \le i \le N\}$, we will use shorthand notations like f(S) for the resulting aggregate value of an aggregate function f. We will also use the notation f(S) to denote the result of repeatedly invoking a scalar function f for all elements of S. We want to emphasize that in this case f(S) denotes a multi-set of values (a new column).

We now define the class of partitionable aggregate functions. An *aggregate* function f is *partitionable for class C* iff two functions f_l and f_g exist such that for any multi-set S and any class C partition S_i of S, $1 \le i \le k$, the following equation holds:

$$f(S) = f_g(\cup_{1 \le i \le k} \{f_l(S_i)\})$$

The notation f_l indicates that the function is applied *locally* (for each partition) whereas f_g is applied *globally.* In addition the result size of the local function f_l must be either bound by a constant or it must be a small fraction of the input size. This requirement is important since otherwise one could use the identity as local function and the sequential aggregation function as the global function. Clearly, this is not desirable since it would not improve processing. In general, the smaller the size of the local results the better the speedup that can be expected, because less data has to be exchanged and there is less input for the global aggregation.

A *scalar* function f is *partitionable for class C* iff a function f_l exists such that for any multi-set S and any class C partition S_i of S, $1 \le i \le k$, the following equation holds:

$$f(S) = \cup_{1 \le i \le k} f_l(S_i).$$

We introduce some notion of extensibility to our communication patterns by parameterizing the partitioning step by means of the partitioning function. We have introduced classes of partitions to allow the optimizer more flexibility w.r.t. to the choice of the partitioning function. The goal is to avoid costly repartitioning of data as often as possible. If only a single (user-defined) partitioning function is allowed for a specific UDF, all data has to be repartitioned before this UDF can be processed. This would result in high processing costs. Thus an improvement is essential and using a special partitioning function (which might be necessary sometimes) should be avoided in general.

An aggregate function f that is partitionable for class C using the two associated functions f_l and f_g can be evaluated in parallel using the following scheme given input multi-set S and a partitioning function p of class C:

1. Partition S in k subsets S_i, $1 \le i \le k$, using p.
   Distribute the partitions to some nodes in the system.

2. Compute $f_l(S_i) = I_i$; for $1 \le i \le k$ for all S_i in parallel.
   Send the intermediate results I_i to a single node for processing of step 3.

3. Compute $f_g(\cup_{1 \le i \le k} \{I_i\}) = f(S)$; f_g can be applied to the I_i in arbitrary order.

Figure 5: Parallel processing scheme for partitionable aggregate functions

Given these definitions the schemes in Figure 5 and Figure 6 show how partitionable functions can be processed in parallel. All k partitions can be processed in parallel. The actual degree of parallelism (i.e. mainly the parameter k) has to be chosen by the optimizer as usual.

For partitionable aggregate functions, f_g can be applied to the intermediate results I_i in arbitrary order in step 3, i.e. pipelining can be used between steps 2 and 3. This allows pipelining parallelism between these steps. Pipelining parallelism will be especially useful in the presence of execution skew (e.g. caused by skewed partition sizes or by unbalanced loads in multi-user mode). The reason is that slightly varying execution times of the local functions do not necessarily lead to an increased overall processing time as the processing of the global function in step 3 can be done in arbitrary order w.r.t. the results of the local functions.

If an aggregate function is used in combination with grouping, the optimizer can also decide to process several groups in parallel. In this case grouping might be done supported by the algorithms described in

[17]. The algorithms discussed there can be applied orthogonally to our framework. Of course, if enough parallelism is possible by processing different groups in parallel, the optimizer might well decide that no further parallel processing of the aggregate function is needed.

A scalar function f that is partitionable for class C using the associated function f_l can be evaluated in parallel using the following scheme given a multi-set S and a partitioning function p of class C:

1. Partition S in k subsets S_i, $1 \leq i \leq k$, using p.
   Distribute the partitions to some nodes in the system.

2. Compute $f_l(S_i)$ for $1 \leq i \leq k$ for all S_i in parallel.

Figure 6: Parallel processing scheme for partitionable scalar functions

Please note, that for the scheme in Figure 6, there is not always a need to combine the local results. Hence, an optional combination step is left out.

Functions that are not partitionable have to be processed sequentially. We observe that aggregate functions that need sorted input are in general not partitionable, since they have to operate on a sorted table as global context. Since only a single aggregated value is produced there is usually no way to fragment the sorted table without loosing this global context. On the other hand such aggregate functions will often be able to use early termination to reduce processing costs. Of course, sorting the input can always be done in parallel. Figure 7 shows the processing scenario for these aggregate functions. A scalar function that needs a sorted input and that is partitionable will normally need a RANGE partitioning with a partial replication, because all other partitioning functions destroy the sorted input property.

An aggregate function f that requires a sorted input can be evaluated using the following scheme given input multi-set S:

1. Sort the input S. This can be done in parallel.

2. Compute f(S) without parallelism (but possibly use early termination).

Figure 7: Parallel processing scheme for non-partitionable aggregate functions f with sorted input

One disadvantage of our framework is that it is not possible in all cases to use one implementation of an aggregate function f for both sequential and parallel processing. Thus the developer might have to implement and register three functions to enable parallel as well as sequential processing of a UDF. However we will see in the next section that many aggregate functions in fact need only one or two functions. If one does not need maximum efficiency for sequential evaluation one can simply use the local and global function for sequential execution, too. But this incurs at least the overhead for the invocation of an additional function. However this additional work for the developer will pay off in all applications that profit from the increased potential for parallelism. Besides that, there seems to be no solution that results in less work for the developer. We will discuss the details of UDF registration in section 4.

It should be observed that the schemes in Figures 5-7 use only communication patterns that are well-known from relational DBMS. The scheme in Figure 7 employs parallel sorting, while the one in Figure

6 uses partitioning. The communication pattern in Figure 5 is the two-step scheme (depicted in Figure 4b), which is used for parallel aggregation in relational systems.

3.3. Validation of the Approach

In this section we will validate our approach by applying the concept to some example UDFs shown in Table 1. The first five functions have well known semantics and need not to be explained. However, we want to emphasize that in an ORDBMS a function like the maximum function `MAX` can be defined for any data type available. For example some RDBMS support time durations as a data type, but no sum function that allows to sum up durations easily. Using our framework such a `sum` function can be defined and executed in parallel, as well.

The other functions we consider may need some explanations. The `Variance` function computes the square of the statistical variance of a set S of sample values v_j given the arithmetic mean m of these values as a constant parameter. The partial sums s_i given here are computed as $\sum_j (v_j - m)^2$; $v_j \in S_i$ (with $S_i \subset S$). The `Nth_largest` function selects the Nth largest element of a sorted set. N is a parameter of the function. Please note that the local functions have to return the *subset* consisting of the N largest elements, while the global function just returns the Nth largest *element* and is identical to the sequential implementation. The `Shape_sum` function is an aggregate function returning the geometric union of shapes like polygons or rectangles (e.g. bounding boxes). The `Closest` function computes one of the nearest neighbors in a set of geometric objects in a common coordinate system for a given point in this coordinate system. The `Most_Frequent` function selects the most frequent element of a multi-set. This function is an example of an aggregate function that is partitionable using a partitioning function of class EQUAL Furthermore local, global and sequential implementation functions are identical.

The next function in Table 1 is an example of a scalar function that requires a global external context but does not compute aggregated values and thus has no input-related context. The `CAST(money)` function casts data of the ADT `money` to the currency `US_Dollar`. This function is arbitrarily partitionable but needs a global external context from the database. If the `CAST` function is invoked with a subtype of `money` (e.g. `Can_Dollar`), then it computes the amount in `US_Dollar`. To do this, the function needs the current foreign exchange rates. The current exchange rates for `US_Dollar` are read from the database when the `CAST(money)` function is called for the first time. The exchange rates are then stored in the global context. This design allows to encapsulate the computation of the amounts in `US_Dollar` in the `CAST` function. This simplifies programming considerably because the use of the foreign exchange rates is transparent for the programmer. For an example application of this function let us assume that the following table `Accounts` has been defined:

```
CREATE TABLE Accounts (No INTEGER, Balance money);
```

Since `money` is an ADT, the values in the `Balance` column of the table `Accounts` can have any type that is a subtype of `money`. To select the sum of all balances in `US_Dollar` one would write the following query (assuming that a function `SUM` has been defined for values of the ADT `US_Dollar`):

```
SELECT SUM(CAST(Balance AS US_Dollar)) FROM Accounts;
```

The `CAST(money)` function as well as the `SUM` function are partitionable using any partitioning function (i.e. class ANY) and can thus be executed in parallel. For the `CAST` function this is based on the fact

Table 1 Characteristics of some UDFs

UDF	*Scalar Function*	*Sorted Input*	*Partitioning Class*	*Sequential Implementation*	*Local Implementation*	*Global Implementation*
MAX	N	N	ANY	max	max	max
MIN	N	N	ANY	min	min	min
SUM	N	N	ANY	sum	sum	sum
COUNT	N	N	ANY	count	count c_i	sum{c_i}
AVG	N	N	ANY	average	sum s_i; count c_i	sum{s_i} / sum{c_i}
Variance	N	N	ANY	variance	sum s_i; count c_i	sum{s_i} / sum{c_i}
Nth_largest	N	N	ANY	Nth largest	N largest (set)	Nth largest
Shape_sum	N	N	ANY	shape sum	shape sum	shape sum
Closest	N	N	ANY	closest	closest	closest
Most_Frequent	N	N	EQUAL	most frequent	most frequent	most frequent
CAST(money)	Y	N	ANY	CAST money	CAST money	-
Rank	N	Y	-	rank	-	-
Median	N	Y	-	scan position	-	-
N_Tile	N	Y	-	N tile	-	-
Cumulative	Y	Y	-	cumulative	-	-
Running_Sum(N)	Y	Y	RANGE (N)	running sum	running sum	-
Running_Avg(N)	Y	Y	RANGE (N)	running avg.	running avg.	-

that there is no input-related context. Thus, if the function is at all partitionable, it can be processed using an arbitrary partitioning. Because the function does not manipulate data inside or outside the database, it has only a “read only” external context. During the first call of the CAST function the exchange rates are read once for each partition on which the function is executed and are then stored in a separate global context for each partition. To provide a better support for functions that have a “read only” external context, it might be useful to provide a special initialization function that retrieves the global context before the function is invoked the first time. In this case one could distribute this global context to all processes that take part in the parallel evaluation. This might be necessary if the result of the retrieval is variant over time. For example the foreign exchange rates vary over time. In this case, since the exchange rates are stored in the database there is no problem if the SQL statement is processed with an SQL isolation level of READ COMMITTED or higher. Please note that the CAST function is the only example of a function with an external context that we discuss. As a general rule all functions without an input-related context, but with an external context can use ANY as a partitioning class, if they can be processed in parallel at all.

The next functions in Table 1 need a sorted input. We have already described their semantics in section 2. Observe that among these functions all aggregate functions are not partitionable and that partitionable scalar functions need a partitioning function of class RANGE including replication of input elements.

This holds for example for the two scalar functions `Running_Avg` and `Running_Sum` that need to see the last N values in their sorted input in order to compute the correct result. It should be noted that these functions need not return a value (not even `NULL`) for replicated input values to guarantee that the same number of result tuples is produced as in sequential processing.

In summary, these examples show that parallel evaluation of many aggregate functions can be supported straightforward using the introduced framework and it's communication patterns, because many aggregate functions actually belong to the class of partitionable functions. All functions shown in table 1 are partitionable or use a sorted input. Thus all example functions can either be executed completely in parallel or use at least parallel sorting.

4. Implementation and Performance Issues

In this section we discuss the issues that are related to the implementation of our framework in ORDBMS and show that our approach can be implemented in existing systems with only minor extensions. Furthermore we give a rough performance analysis of parallel processing for partitionable functions in terms of the possible speedup.

4.1. Implementation and Integration in existing ORDBMS

We assume an ORDBMS implementation having an usual query execution framework that processes aggregate functions (in combination with GROUP-BY) using one or two operators [16]. All operators are assumed to process their input streams one object at a time. Parent and child operators of the UDF operator may be arbitrary other operators in our framework. We assume an operator model for parallel execution of database operators [16], i.e. there exists a communication operator that allows to ship streams of objects between different execution units (maybe on different nodes) of the parallel execution engine and thus isolates the operator implementations to a large portion from their parallel evaluation.

Please note that there are no general restrictions of our approach to a certain DBMS architecture (shared-everything, shared-nothing, data-sharing). The differences between the various architectures that are relevant to our approach should be mainly encapsulated by the communication mechanism. In addition the cost parameters that are used by the optimizer are affected. This can result in different degrees of parallelism on different architectures, but does not impact our processing framework fundamentally. Pipeline parallelism for operator chains can be applied, but is no essential requirement.

Prior to processing a user-defined function, the associated functions have to be registered with the ORDBMS. Figure 8 shows the extensions we propose to the UDF registration as discussed in section 2.

We have marked our extensions in boldface. The ORDER BY clause can be used to specify a sort order that is required for the input table, on which the function is executed. The external local and global function parameters are needed to specify the functions that have to be used for parallel evaluation, if a new aggregate function is registered. For scalar functions the external local function is used for parallel evaluation of data partitions. To enable parallel evaluation, the partitioning class has to be specified (one could also use ANY as a default value). In addition to ANY, EQUAL and RANGE partitioning, the developer can register a specific (user-defined) partitioning function for a UDF. In the latter case, the data will almost always be repartitioned for the parallel evaluation of this UDF. Of course the various functions that are registered must be consistent w.r.t. their argument types. For example the sequential and the local functions must have the same argument types (but will often have different return types).

Please note that additional information about these functions should be supplied by the developer.

```
CREATE [FUNCTION | AGGREGATE] <name> (<argument type list>)
RETURNS <data type name>
EXTERNAL NAME <external function name>
[ORDER BY {<argument name> USING <sort function name> [ASC | DESC] }]
[EXTERNAL LOCAL NAME <external function name>]
[EXTERNAL GLOBAL NAME <external function name>]
[PARTITIONING CLASS ( ANY | EQUAL (<argument name>)
     | RANGE (<argument name>, <argument name or constant>)
       USING <sort function name> [ASC | DESC]
     | <partitioning function name>)]
LANGUAGE <language name>
...
```

Figure 8: Extensions to UDF registration

Such information should include the usual cost parameters. In addition information about the size of the intermediate results of the local aggregation function (depending perhaps on the cardinality of the input set, if the function returns a collection type) should be included. These informations can be provided to the system using the normal interfaces provided for UDFs. For example, in a system that uses the system catalog to store cost parameters this requires only minor extensions. In addition to the registration of the new UDFs itself, support functions like special partitioning or sort functions have to be registered themselves.

It has been observed [24] that it may be of general interest to allow the registration of more than one function that implements a UDF due to efficiency reasons. This enables an object-relational optimizer to choose the implementation that has the least cost for the specific query at hand, much like it is practised by today's relational optimizers w.r.t. to the standard RDBMS operators like joins. This feature is also of interest for multi-representational data types (see e.g. Illustra), i.e. data types that have a different physical representation depending e.g. on the size of the data object. Aggregate functions might also profit from different implementations that are appropriate for different input set cardinalities. We have seen here that different implementations are already often needed for sequential and parallel evaluation.

The optional sort requirements can be integrated into rule-based query optimization (see e.g. [18], [36], [37], [38]) simply by specifying the sort order as a required physical property for the operator executing the UDF. Then a sort enforcer rule can guarantee this order requirement by putting a sort operation into the access plan if necessary. Informix' Illustra [23] supports already optional sorting of inputs for UDFs that have two arguments and return a boolean value. The developer can specify a user-defined order for the left and right input of such a function. Obviously this allows to implement a user-defined join predicate using a sort-merge join instead of a cartesian product followed by a selection. Thus our proposal can be seen as an extension of this approach w.r.t. to a broader class of supported UDFs and their parallel execution.

Implementing RANGE partitioning is a bit complicated since a user-defined sort order and partial replication have to be supported. One difficulty is e.g. to find equally populated ranges for a given user-defined sort-order. We believe that range partitioning with partial replication can be best supported by an appropriate extension of the built-in sort operator of the ORDBMS. This sort operator has to support user-defined sort orders anyway. The definition of ranges and partial replication can be supported, if information about the data that is sorted is collected during the sort operation.

The exact implementation depends on the concrete algorithm that is used for the sort operator. In addition to that extension, the operator that invokes UDSFs has to be extended slightly. The UDSF that needs the range partitioning is evaluated immediately after the partitioning. Replicated data elements (that have to be marked) are processed by the UDSF in a special mode that has to be indicated by turning a special switch on. In this mode only the global context of the UDSF is initialized and no results are produced. For example, when computing the function `Running_Avg` over five values, the first four values of a partition will be replicated ones and are stored in the global context of the function. Then, the fifth invocation produces the first result.

We want to emphasize again that the query optimizer can try to avoid data repartitioning, when multiple UDFs are processed, if the developer specifies only the class of the partitioning functions. This can reduce processing costs dramatically, especially for shared-disk and shared-nothing architectures. If the developer has to specify a single (user-defined) partitioning function for each UDF, in almost all cases a repartitioning step will be needed to process a UDF. Vice versa, if a single partitioning function satisfies all of the occuring partitioning classes of the current set of UDFs, then repartitioning w.r.t. UDFs can be avoided.

4.2. Performance Discussion

We will now discuss the possible performance benefits for processing partitionable aggregate functions in our framework. Note that this analysis is straightforward for partitionable UDSFs. We use the obtainable speedup of the parallel execution of a UDAF to estimate the efficiency of our approach. It has to be emphasized that we compare the sequential and parallel execution of the aggregation function only, i.e. we do not try to estimate the speed-up for the evaluation of complete queries.

For our analysis we use a simple analytic cost model that takes only CPU costs into consideration. This seems to be roughly appropriate to us as the aggregated size of main memories tends to increase on a parallel system compared to a uniprocessor system. Thus the necessary I/O will almost always be less in the parallel case. CPU costs for communication can be quite significant for parallel evaluation, but will vary to a great extent depending on the architecture used. For example in shared-everything the costs for exchanging intermediate results between processors are negligible, if shared memory is used properly. We do not expect that communication bandwidth will be a problem given the rapid progress in networking technology (consider e.g. a switched giga bit ethernet or an ATM based environment). Therefore we will also ignore bandwidth problems. The costs of computing the UDAF f sequentially may be different from the costs when using f_g and f_l for sequential evaluation, but in order to keep our model simple we assume that these costs are nearly equal (i.e. cost (f_l) + cost $(f_g) \approx$ cost (f)). Thus our cost model has the following three components:

1. F: CPU cost for evaluating the UDAF. $F = \text{cost}(f_l) + \text{cost}(f_g) \approx \text{cost}(f)$

2. P(d): CPU cost for data partitioning depending on the degree of parallelism d, i.e. the number of different nodes that process a partition.

3. I(d): CPU cost for exchange of intermediate results depending on the degree of parallelism d.

The uniprocessor cost for the processing of the UDAF is simply F and the costs of parallel processing are F + P(d) + I(d). When using parallel query evaluation the input streams will often be already partitioned on different execution units and without the possibility to evaluate the UDAF in parallel all the data has to be shipped to a single execution unit. Therefore the cost P(d) will be low in most cases when compared to a sequential evaluation of the UDAF. I(d) depends on the size of the sub-aggregates and thereby on the implementation of the specific UDAF at hand. If the size of the intermediate results is low compared to the size of the input set the relative costs I(d) are quite low. Thus the obtainable speedup SP (depending on the degree of parallelism d) can be approximated roughly as follows:

$$SP(d) = \frac{\text{sequential execution time}}{\text{parallel execution time}} = F/\ [(F + P(d) + I(d))/d] \approx d*F/\ F = d$$

This approximation is especially applicable if the UDAF is expensive, i.e. if $F >> P(d) + I(d)$.

In summary, we have seen that a near linear speedup can be expected in many cases. Thus our framework for parallel processing can be of great advantage for the overall query processing time. Whether or not parallelism for UDAFs should be used for a given query, is a cost-based decision the optimizer has to make. The same is also true for partitionable scalar functions.

5. Related work

User-Defined Functions (UDFs) have attracted increasing interest of researchers as well as the industry in recent years (see e.g. [19], [11], [15], [13], [12], [4], [24]). Despite this, most of the work discusses only the non-parallel execution of UDFs. In [13] pipeline parallelism for functions as well as intra-function parallelism are discussed. Intra-function parallelism allows a single function invocation to be processed in parallel. This can be useful for extremely expensive functions, e.g. a function processing a large data object like a satellite image. All concepts seem to be orthogonal to our framework, but only applicable for scalar functions.

In [16] and [17] parallel processing of aggregate functions in RDBMS has been studied. The proposed concepts are applicable for built-in aggregation functions and consider also aggregation in combination with GROUP-BY operations and duplicate elimination. It has been observed [16] that different local and global functions are needed for parallel aggregation operations in RDBMS. The proposed algorithms in [17] may be combined with our framework, if user-defined aggregate functions are used with GROUP-BY. To the best of our knowledge there is no work dealing specifically with parallel processing of user-defined aggregate functions. We also do not know any other concepts for parallel processing of scalar user-defined functions that have a global context. In [34] RDBMS are extended by ordered domains, but neither is an object-relational approach taken nor are functions considered.

It is interesting to compare our classification of aggregate functions in partitionable and non-partitionable aggregate functions with other classifications. In [21] a classification of aggregate functions into three categories is developed primarily with the goal to be able to determine if super-aggregates in data cubes can be computed based on sub-aggregates for a given aggregate function. It is also pointed out that this classification is also useful for the parallel computation of user-defined aggregate functions. In the proposed classification an aggregate function f with a given input multi-set S and an arbitrary partition S_i of S is:

1. *distributive* iff there is a function g such that $f(S) = g(\cup_{1 \leq i \leq k} f(S_i))$.

2. *algebraic* iff there is an M-tuple valued function g and a function h such that $f(S) = h(\cup_{1 \leq i \leq k} g(S_i))$. It is pointed out that the main characteristic of algebraic functions is that a result of fixed size (an M-tuple) can summarize sub-aggregates.

3. *holistic* iff there is no constant bound on the size of the storage needed to represent a sub-aggregate.

Clearly, distributive and algebraic functions are both partitionable aggregate functions for the partitioning class ANY. Note that our definition of partitionable aggregate functions is less restrictive w.r.t. the size of the sub-aggregates. Aggregate functions that are easy to implement using a sorted input are typically holistic. Aggregate functions that are partitionable with a less general partitioning class than ANY, e.g. the `Most_Frequent` function, are holistic in this scheme, but can be evaluated in parallel. Note that the application scenario in [21] is different to ours w.r.t. to partitioning and parallel evaluation, because the sub-aggregates in data cubes must be computed for fixed partitions that are derived from semantically predefined sub-cubes and not by the application of an arbitrary partitioning function. But of course parallel aggregation is also useful for computing those aggregate values in a data cube that have to be determined directly from the base tables (e.g. the fact table in a data warehouse).

The classification of Gray et al. was designed with the goal to compute data cubes efficiently. However, the rationale behind our work was to find a classification of functions that is useful for parallel evaluation. We have seen that the central question for our kind of classification is to define which communication patterns should be allowed or supported for parallel execution, and in that context a classification of partitioning functions is indispenseable.

In [22] the class of decomposable aggregate functions is introduced to characterize the aggregate functions that allow early and late aggregation as a query optimization technique. This class of aggregate functions is identical to partitionable aggregate functions of partitioning class ANY except that no size restriction for sub-aggregates is required in [22]. Thus for these partitionable functions also certain rewrite optimizations are possible that provide orthogonal measures to improve query execution performance.

6. Conclusions and Future Work

In this paper we have proposed a framework that enables parallel processing of a broad class of user-defined functions in ORDBMS. This removes a performance bottleneck in parallel object-relational query processing.

Since it was clear that a straightforward application of data parallelism (as known from parallelizing relational operators) is not possible, we had to devise more sophisticated parallelization techniques that are applicable for a broad class of UDFs. As we have shown, this task involved two key issues: first, we had to be concerned with extensibility w.r.t. to communication patterns in the execution engine of parallel ORDBMS. Second, we had to define new interfaces that allow the developer to teach the database system more about the semantics of UDFs. This can be seen as a step towards making the “black box” UDFs of current ORDBMS less opaque and thus allow more sophisticated query optimization and more efficient execution. In our view truly object-relational query processing requires still many more moves in the direction of such a classification-based processing.

We have solved the first task referring primarily to the commonly used communication patterns of parallel RDBMS. We have defined only one new communication pattern for UDFs with an external global context. We believe that providing a fixed set of built-in communication patterns is the right approach. Allowing extensibility in the sense of user-defined communication patterns seems to be not feasible to us, as this would lead to an overwhelming complexity both in the executor and optimizer. Instead we proposed extensibility (for communication patterns) at the level of partitioning functions to enable the developer to flexibly use the built-in communication patterns. As we have shown, a classification of partitioning functions is necessary to avoid costly data repartitioning operations as much as possible. Our approach to the second task was to introduce a classification of UDFs that in addition to functional properties also includes implementation characteristics. Based on that classification we came up with generally applicable parallel processing schemes that we further optimized using e.g. techniques like early termination. Introducing sorting as a preprocessing step allows for an easier implementation of UDFs, increases the semantic power, and enables the use of parallelism in the preprocessing phase. In conclusion our framework fits well into existing ORDBMS implementations and keeps the complexity for the developer in reasonable bounds.

Some important questions for our framework and for parallel object-relational query processing in general are:

• Are there other classes of UDFs that do not comply with our methodology? As an example consider user-defined table functions [25] that can e.g. be used to encapsulate access to external data sources or external indexes [25].

• Is the extensibility of our framework sufficient for all classes of UDFs? We believe that in the future, when ORDBMS are maturing, additional communication patterns might emerge in close cooperation between vendors, developers and the research community.

• Though our framework supports parallel execution of user-defined predicates, we believe that substantial additional work is necessary to avoid cartesian product operations in ORDBMS that are used to deal with most user-defined join predicates. We are currently working on a framework for user-defined join algorithms to overcome cartesian products and allow efficient parallel execution.

• How can parallelism be used to efficiently process UDFs on single but very large ADTs [13] and collection types in parallel ([8], [35])? We have not addressed these important questions here.

Additional future work should be concerned with the impact of user-defined aggregate functions and the proposed sorting requirements on query optimization. Another interesting question is how materialized user-defined aggregates can be maintained at low cost, e.g. for being used in a data warehouse (see [21]) application.

7. References

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

[2] Gray, J.: A Survey of Parallel Database Techniques and Systems, in: Tutorial handout at Int. Conf. on Very Large Databases, 1995.

[3] Valduriez, P.: Parallel Database Systems: Open Problems and New Issues, in: Distributed and Parallel Databases, Vol.1, No. 2, April 1993, pp.137-166.

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

[5] Carey, M. J., Dewitt, D.J.:Of Objects and Databases: A Decade of Turmoil, Proc. of the 22nd VLDB Conference, Mumbai (India), 1996.

[6] Chamberlin, D.: Using the New DB2, Morgan Kaufman Publishers, San Francisco, 1996.

[7] Davis, J. R.: Creating an extensible, Object-Relational Data Management Environment: IBM's Universal Database, White Paper, Database Associates International,1996.

[8] DeWitt, D.: Parallel Object-Relational Database Systems: Challenges & Opportunities, invited talk, PDIS 1996.

[9] DeWitt, D. J., Carey, M., Naughton, J., Asgarian, M., Gehrke, J., Shah, D.: The BUCKY Object-Relational Benchmark, to appear, Proceedings of the 1997 SIGMOD Conference, Tucson, Arizon, May, 1997.

[10] Mattos, N.: An Overview of the SQL3 Standard, Database technology Institue, IBM Santa Teresa Lab, San Jose, California, July 1996

[11] Mattos, N., Dessloch, S., DeMichiel, L., Carey, M.: Objekt-Relational DB2, IBM White Paper, July 1996.

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

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

[14] 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, to appear, Proceedings of the 1997 SIGMOD Conference, Tucson, Arizon, May, 1997.

[15] Stonebraker, M., Moore, D.: ORDBMS - The next Great Wave, Morgan Kaufman Publishers, 1996.

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

[17] Ambuj Shatdal, Jeffrey F. Naughton: Adaptive Parallel Aggregation Algorithms. SIGMOD Conference 1995: 104-114.

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

[19] Michael Stonebraker: Inclusion of New Types in Relational Data Base Systems. ICDE 1986: 262-269

[20] Harinarayan, V., Rajaraman, A., Ullman, J.D.: Implementing Data Cubes Efficiently, Proceedings of the 1996 SIGMOD Conference, Montreal, Canada, 1996.

[21] Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., Pirahesh, H.: Data Cube: A Relational Aggregation Operator Genreralizing Group-By, Cross-Tab, and Sub-Totals, Data Mining and Knowledge Discovery 1, p. 29-53, Kluwer Academic Publishers, 1997.

[22] Weipeng P. Yan, Per-Åke Larson: Eager Aggregation and Lazy Aggregation. VLDB 1995: 345-357

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

[24] Praveen Seshadri, Miron Livny, Raghu Ramakrishnan: The Case for Enhanced Abstract Data Types. VLDB 1997: 66-75

[25] Stefan Deßloch, Nelson Mendonça Mattos: Integrating SQL Databases with Content-Specific Search Engines. VLDB 1997: 528-537

[26] Michael Stonebraker: The Case for Shared Nothing. Database Engineering Bulletin 9(1): 4-9 (1986)

[27] Oracle Corporation, http://www.oracle.com/st/cartridges/time/, August 1997.

[28] Oracle Corporation, http://www.oracle.com/st/, August 1997.

[29] Oracle Corporation, http://www.oracle.com/st/cartridges/context/, August 1997.

[30] Informix Corporation, http://www.informix.com/informix/products/techbrfs/dblade/program/2122871.htm, August 1997.

[31] Red Brick Systems, Inc., http://www.redbrick.com/rbs-g/html/whpap.html, August 1997.

[32] Joseph M. Hellerstein, Jeffrey F. Naughton, Avi Pfeffer: Generalized Search Trees for Database Systems. VLDB 1995: 562-573.

[33] Wayne Niblack, Ron Barber, William Equitz, Myron Flickner, Eduardo H. Glasman, Dragutin Petkovic, Peter Yanker, Christos Faloutsos, Gabriel Taubin: The QBIC Project: Querying Images by Content, Using Color, Texture, and Shape. Storage and Retrieval for Image and Video Databases (SPIE) 1993: 173-187.

[34] Wilfred Ng, Mark Levene: OSQL: An Extension to SQL to Manipulate Ordered Relational Databases. IDEAS 1997: 358-367.

[35] Michael J. Carey, Nelson Mendonça Mattos, Anil Nori: Object-Relational Database Systems: Principles, Products, and Challenges (Tutorial). SIGMOD Conference 1997: 502 .

[36] Guy M. Lohman: Grammar-like Functional Rules for Representing Query Optimization Alternatives. SIGMOD Conference 1988: 18-27.

[37] Laura M. Haas, Walter Chang, Guy M. Lohman, John McPherson, Paul F. Wilms, George Lapis, Bruce G. Lindsay, Hamid Pirahesh, Michael J. Carey, Eugene J. Shekita: Starburst Mid-Flight: As the Dust Clears. TKDE 2(1): 143-160 (1990).

[38] William J. McKenna, Louis Burger, Chi Hoang, Melissa Truong: EROC: A Toolkit for Building NEATO Query Optimizers. VLDB 1996: 111-121.

Appendix: an Example Application of an User-Defined Aggregate Function using Sorting

In this section we will describe the application of a UDAF that requires ordered input to implement a new feature for an existing document mangement system called ODS. We will first describe ODS and the profiling service as a new application to be implemented on ODS. We then show how our concepts can be applied to allow an efficient parallel processing even supporting multi-query optimization techniques for the processing of profiles.

ODS provides a boolean retrieval model answering queries on the existence of word patterns, words, phrases, and boolean expressions of them in documents. Syntax and semantics of the query language can be informally described as follows:

`<word>`	`Exact match of <word>`
`%`	`Wild card character usable in a <word>, allowing to specify simple pattern matching`
`.`	`Distance operator, used to combine terms to phrases`
`&`	`Boolean AND of phrases`
`|`	`Boolean OR of phrases.`

A *term* is defined as a word or a word pattern. Terms connected by distance operators form *phrases*. This simple language defines the basis of the retrieval service, as for example, the phrase “deduct% . database” asks for all documents in which there is an occurrence of a word starting with the string ‘deduct’ together with a word ‘database’ and having at most one other arbitrary word in between as indicated by the distance operator.

The document retrieval model of ODS is based on full-text indexing that, in the case of ODS, refers to plain relational tables, which are organized in such a way that document retrieval can be done efficiently. There are two important tables, a table *WORDS* (*word*, *Word-ID*), which maps each word which appears in at least one document to a numeric key *Word-ID*, and a table *DOCUMENTS* (*Word-ID*, *Doc-ID*, *Pos*), which maps word keys to numeric document keys *Doc-ID* and position numbers *Pos* depending on where the word appears in the documents; primary keys of the tables are underlined. A query that is specified as a boolean retrieval expression is mapped to an SQL query that uses the full-text index tables to search for all document keys that match the given expression. These queries can be quite complex. In reality the full-text tables are more complex and optimized but these details are not of interest here.

We will now define our new application, the profiling service. Basically, a profile represents the user’s reading interests and can be expressed as a single full-text retrieval query in ODS. The profiling service is the (batch) execution of a large set of profiles, which can be launched in fixed time intervals, for example once per night. A sample profile, also used in the following, may look like this:

```
(‘deduct% . database | multimedia database’)
```

Please observe that profiles are usually evaluated in batch mode, which provides opportunities for multi-query optimization, i.e. to process a set of queries in an integrated, tightly coupled and optimized fashion. Obviously it is a great advantage if one evalutes common sub-expressions that occur in different profiles only once. Common sub-expressions can be identified by searching for identical terms. This analysis can be done off-line and the result can be stored in three tables shown in Figure 9. Each term that ocurrs in a profile, i.e. each word or word pattern, is assigned a term identifier *(Term-ID)* and stored in the *TERMS* table. Terms connected by distance operators and boolean AND operators form *subprofiles*. In our example, there are 3 subprofiles. As the number of terms in a subprofile is variable, we have chosen to describe a subprofile in the *SUBPROFILES* table in the following way: for each term of a subprofile a corresponding tuple is inserted; the last column, *ProfilePos*, holds the position of the term within its subprofile; the *Distance* column expresses the number of words between the actual term and the next term in the subprofile as is defined by the distance operator used in that subprofile. The special value -1 of the *Distance* column expresses that there was no distance specified, and the value -2 illustrates that there are no further terms in the subprofile. Finally, the *PROFILES* table represents the profiles in disjunctive normal form over subprofiles. The tables store the description of the profiles and have to be updated if the set of profiles changes. Please note that common search terms occur only once in the *TERMS* table. This reduces the costs of searching considerably.

Given these tables, how can we evaluate the profiles? Clearly, writing one SQL query that does the complete work would be desirable. Unfortunatly this is not possible without user-defined extensions. The reason is that it is not possible to find the documents that match a subprofile in traditional SQL. This is due to the fact that subprofiles define patterns consisting of a varying number of terms. In addition the terms have to occur in a certain order and within certain distances in a document. If user-defined aggregate functions are allowed, the evaluation of all profiles can be easily expressed as a single query. In addition, if the user can specify a required order for a function the implementation of the respective user-defined aggregate function becomes easy. Call such a function `Match_SubProfile`. The function returns the value 1 when a document matches a given subprofile and 0 otherwise. The following query does the evaluation using the full-text index tables *WORDS* and *DOCUMENTS*:

```
SELECT DISTINCT Profile-ID, Doc-ID
FROM Profiles,
     (SELECT SubPro-ID, Doc-ID
     FROM Terms, Subprofiles, Words, Documents
     WHERE   Words.Word LIKE Terms.Term AND
             Words.Word-ID = Documents.Word-ID AND
             Subprofile.Term-ID = Terms.Term-ID
     GROUP BY SubPro-ID, Doc-ID
     HAVING Match_SubProfile(Pos, ProfilePos, Distance) = 1)
     AS S
WHERE P.SubPro-ID = S.SubPro-ID;
```

The `Match_SubProfile` aggregate function is called with the set of all positions in a certain document that contain a word matching a term of a certain subprofile. If the requirement “ORDER BY Pos” (i.e. the first argument) is specified, the function is simple to implement. It scans the document positions in order and checks whether there is a sequence of inputs having consecutive profile positions starting with value 1 and ending with value -2. In addition, the specified distances between the positions must be correct. If at least one such sequence exists, the function finally returns a 1 otherwise a 0. This allows to enhance the processing by using the early termination feature introduced in section 3.1. Therefore the `Match_SubProfile` function is an example of an aggregate function that can use a sorted input and early termination. As a result of the application of the `Match_SubProfile` function, the table expression S returns one tuple containing a Subprofile-ID and a Doc-ID for each document and subprofile pair, if the document matches the expression of the subprofile. Finally, in the outer query block all matches found are joined with the *PROFILES* table. This results in a list of profiles and the matching Doc-IDs.

Please observe that all profiles are now evaluated with a single query. This query can be processed completely in parallel with the `Match_SubProfile` function being the only exception. But this function can be computed in parallel for all groups. In addition the tuples in each group could be sorted in parallel if the groups are very large. This allows parallel processing of the complete profile evaluation in a way that is transparent for the application. Without the possibility to use a user-defined aggregate function, the profile evaluation would have to be done using embedded SQL. This would have resulted in much less parallelism as only a part of the table expression S could have been processed by the following SQL query:

```
SELECT SubPro-ID, Doc-ID
FROM Terms, Subprofiles, Words, Documents
WHERE    Words.Word LIKE Terms.Term AND
     Words.Word-ID = Documents.Word-ID AND
     Subprofile.Term-ID = Terms.Term-ID
ORDER BY SubPro-ID, Doc-ID, Pos
```

This embedded SQL approach results in a poor performance and response time as expensive parts of the evaluation have to be done in the application.

This example application of user-defined set functions shows that application logic can be pushed from the application programming level down into the deeper levels of the database system. This allows programming with high productivity and substantial performance gains, especially if parallel database server technology is available for evaluation.

Table *TERMS*

Term	*Term-ID*
architect%	6
data%	4
database	2
deduct%	1
multimedia	3
system	5

Table *SUBPROFILES*

SubPro-ID	*Term-ID*	*Distance*	*ProfilePos*
1	1	1	1
1	2	-2	2
2	3	0	1
2	2	-2	2
3	4	-1	1
3	5	0	2
3	6	-2	3

Table *PROFILES*

Profile-ID	*SubPro-ID*
1	1
1	2
2	2
2	3

Figure 9: Profile Representation for Two Sample Profiles
(primary keys are underlined, common expressions are shaded):
Profile 1: ‘deduct% . database | multimedia database’
Profile 2: ‘multimedia database | data% & system architect%‘