On Parallel Processing of Aggregate and Scalar Functions in Object-Relational DBMS

Michael Jaedicke, Bernhard Mitschang
Technische Universität München
Computer Science Department
80290 München, Germany
+49 89 48095 184
jaedicke@informatik.tu-muenchen.de

1. 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. Thus one of the current goals for parallel object-relational DBMS is to move towards higher performance. In this paper we develop a framework that allows to process user-defined functions with data parallelism. We will describe the class of partitionable functions that can be processed parallelly. We will also propose an extension which allows to speed up the processing of another large class of functions by means of parallel sorting. Functions that can be processed by means of our techniques are often used in decision support queries on large data volumes, for example. Hence a parallel execution is indispensable.

1.1 Keywords

User-defined functions, aggregates, parallel query processing, object-relational database systems

2. INTRODUCTION

According to [42] object-relational database management systems (ORDBMS) are the next great wave. 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 [30], geographic information systems [36], and management of time series [35] and documents [34]. 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 of some database vendors ([7], [32], [31], [33]). Although first industrial implementations enter the marketplace and the SQL3 standard [26] is maturing, there are still many topics left for research in this area ([10], [11], [2], [3]).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD '98 Seattle, WA, USA
© 1998 ACM 0-89791-995-5/98/006...$5.00

One of the current goals for ORDBMS is to move towards a framework for constructing parallel ADTs [3] and more sophisticated query optimization and execution ([3], [38]). 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 enables more sophisticated query optimization and execution as the ORDBMS knows and understands at least part of the semantics of the ADT. This will result in great performance improvements ([42], [36]). While there are different approaches to reach this goal ([22], [4], [38]), most ORDBMS vendors currently offer ADT developers some parameters to describe the semantics of user-defined functions.

Our main contribution in this paper is to show how a broad class of user-defined functions can be processed parallelly. This class includes user-defined aggregate functions - often called set or column functions. 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 computing 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 [39]. In addition, if scalar functions with a global context are processed parallelly, caution is needed in order to get semantically correct results. Our framework can help in this case, too. Furthermore, we show that many aggregate functions can be easily implemented, if their input is sorted out, and they can thus profit from parallel sorting.

In Section 3 we give the necessary background on user-defined functions and show their limits with respect to

[1] This work was partially supported by the DFG (SFB 342, B2).

parallel execution. Our framework for parallel processing of user-defined functions is introduced in Section 4. After a discussion of related work in Section 5, the closing Section 6 contains a short summary and a brief outlook to future work.

3. USER-DEFINED FUNCTIONS

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 ([9], [14], [12], [43], [41]) and object-relational query processing ([42], [4]).

3.1 Built-in 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 a single row of an input table.

By 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. 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 [4].

In ORDBMS it is possible to use a *user-defined* function (UDF) in 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).

3.2 Definition of New UDFs

Let us now describe briefly how UDFs are created in ORDBMS. Users can write UDFs as so-called *external* functions in a 3GL (typically C and Java are supported as languages) and then register them with the DBMS.

In advanced object-relational systems it should be possible to implement the body of UDFs using SQL statements embedded in the code of a 3GL (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. Furthermore a UDF 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 ORDBMS already supports UDFs that consist of one or more SQL statements.

If DML statements or external actions are used in UDFs, the UDF might depend on arbitrary data in the database or elsewhere. We say that these functions have an *external* context. Since we do see only very limited opportunities for database technology to enable parallelism for functions with external context we will not consider that kind of UDFs in this paper.

3.2.1 User-Defined Scalar Functions

Figure 1 provides an example of the syntax used in DB2 UDB [4] to register a new UDSF with the DBMS. The scalar function `add` returns the sum of its two arguments of the user-defined data type `dollar`.

```
CREATE FUNCTION add (dollar, dollar)
RETURNS dollar
EXTERNAL NAME 'dollar!add'
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 UDSF `add` 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 [4] for most of the details and provide only the relevant information to our problem. An interesting feature is the possibility to use a so-called scratchpad area for UDSFs. A scratchpad area is a small piece of memory that is passed to a UDSF with all calls and that is not deleted after the executed function returns the control. Thus it is possible for a function to maintain a global context (or global state) that means 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. Often a scratchpad is used to store intermediate results that have been computed from the arguments of former function calls. We say that such UDSFs have an *input* context. The moving average function is an example of such a scalar function. It allows to compute many aggregates (many moving averages) by means of a single scan over an input table.

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 that is read by the function. In addition to this, the percentage of the argument's storage size that is processed at the average call to the UDF 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 [42]. Since providing these details can be a time consuming task, easy to use development kits may be offered for this task ([7], [23], [33]).

3.2.2 User-Defined Aggregate Functions

Let us now see how, for example, the Informix Illustra ORDBMS supports UDAFs. The system computes aggregate functions 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 external functions to implement a new UDAF:

- *Init*():
  The Init function is called only once and without arguments to initialize the aggregate computation before the actual computation of the aggregate begins. It returns a pointer to memory, which it has allocated to store intermediate results during the aggregation.
- *Iter*(pointer, value):
  The Iter function is called once for each element of the input set. One parameter is the value of this element and the other is the pointer to the allocated memory. It aggregates the next value into the current aggregate that is stored in memory using the pointer. It returns the pointer to the allocated memory.
- aggregate value = *Final*(pointer):
  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 using the pointer to the allocated memory. In addition, it deallocates the memory.

The pointer, similar to the scratchpad area mentioned before (cf. [4]), allows to store the input context of the computation. For example 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 allocated memory. The *Final* function would divide the sum by the number and return the result. The reader should note that all practical aggregate functions have an input context.

Obviously this design matches the usual Open-Next-Close protocol [12] for relational operators. 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.

3.3 Limits of Current ORDBMS

We will now describe the limits of current ORDBMS with respect to the parallel execution of UDFs. To provide a concrete example, we use the user-defined aggregate function `MOST_FREQUENT`, which computes the most frequently occurring integer value in a column of type integer. We have omitted some details to make the presentation as simple as possible.

We first have to create three UDSFs `INIT_MF`, `ITER_MF`, `FINAL_MF` that provide the implementation routines of the `MOST_FREQUENT` aggregate function. These three routines are programmed as external functions, i.e. they are written e.g. in C and can use the system-provided API for UDFs to handle tasks like memory allocation, etc. Then they are registered using the `CREATE FUNCTION` statement:

```
CREATE FUNCTION INIT_MF()
RETURNS POINTER
EXTERNAL NAME 'libfuncs!mf_init'
LANGUAGE C ...;

CREATE FUNCTION ITER_MF(POINTER, INTEGER)
RETURNS POINTER
EXTERNAL NAME 'libfuncs!mf_iter'
LANGUAGE C ...;

CREATE FUNCTION FINAL_MF()
RETURNS INTEGER
EXTERNAL NAME 'libfuncs!mf_final'
LANGUAGE C ...;
```

The function `INIT_MF` allocates and initializes memory to store the integer values together with a count and returns a pointer to that memory. The function `ITER_MF` stores its argument in the allocated memory, if it is an integer value not seen so far, and increments the count for this value. Finally, the `FINAL_MF` function searches for the value with the maximum count and returns this value. Now we create the UDAF with the `CREATE AGGREGATE` statement:

```
CREATE AGGREGATE MOST_FREQUENT
(
init = INIT_MF()
iter = ITER_MF(POINTER, INTEGER)
final = FINAL_MF(POINTER)
);
```

Now the `MOST_FREQUENT` function can be used as a new aggregate function in queries. We will now explain, why this aggregate function cannot be processed parallelly.

UDSFs without context can be executed parallelly using data parallelism. Instead of executing a set of function invocations in a sequential order, one simply partitions the data set (horizontal fragmentation) and processes the UDSF for each data partition parallelly. This parallel execution scheme is shown in Figure 2 for a selection.

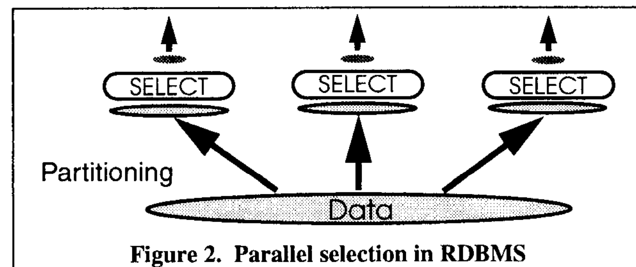

Figure 2. Parallel selection in RDBMS

Obviously aggregate functions cannot use this approach without modification as they have an input context and deliver only a single result for a set of input tuples. Parallel aggregation operations in RDBMS use an execution scheme consisting of two steps [12] as shown in Figure 3. 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 (merging step in Figure 3). For the aggregate function COUNT the local aggregation counts while the global aggregation computes the sum of the local counts. Generally speaking, the local and global aggregation functions needed for parallel execution are different from the aggregate function that is used for sequential execution. For built-in aggregate functions local and global aggregation functions are system-provided. Thus the DBMS can use these functions for parallel execution. For UDAFs there is currently no possibility to register additional local and global aggregation functions. This is the reason, why a UDAF like the MOST_FREQUENT function cannot be executed with the usual 2 step parallel aggregation scheme.

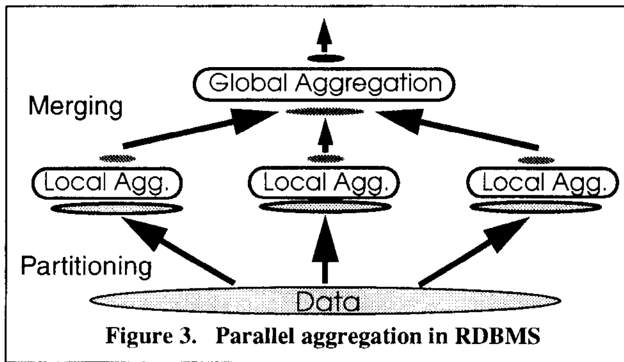

Figure 3. Parallel aggregation in RDBMS

Another problem is that current ORDBMS do not allow the developer to define a special partitioning function for a UDAF. However, unfortunately not all UDAFs can be processed parallelly on all kinds of partitions as we will show later. The latter is also valid with respect to scalar functions that have an input context. In many cases the result will be semantically incorrect if the data partitioning does not take the semantics of the function into consideration.

In summary, UDSFs with an input context and all practical UDAFs can not be processed parallelly without special support by the DBMS. This situation will result in a performance bottleneck in parallel 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 correctly process a UDF with input context, i.e. sequentially, and afterwards the data possibly has to be redistributed for further parallel processing. This results in additional communication costs and hence even worse performance.

4. PARALLEL PROCESSING OF UDFS

In this Section we describe several orthogonal approaches to enhance the parallel processing of UDFs with an input context. In Subsection 4.1 we introduce local and global aggregation functions for UDAFs as a generalization of the relational processing scheme. In Subsection 4.2 we introduce partitioning classes and define the class of partitionable functions that can be processed with data parallelism. In Subsection 4.3 we propose sorting as a preprocessing step to enhance parallel execution for non-partitionable UDAFs.

4.1 Two Step Parallel Aggregation of UDAFs

In this Subsection we will show how aggregates can be processed in 2 steps using local and global aggregate functions.

To simplify the presentation below, we will omit constant input parameters to UDFs. Given a set S, we will use shorthand notations like f(S) for the resulting aggregate value of an aggregate function f applied to S. 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).

Next, we define the class of aggregate functions that can be processed parallelly using local and global aggregation functions. An *aggregate* function f is *partitionable* iff two aggregate functions f_l and f_g exist, such that for any multi-set S and some partition S_i of S, $1 \leq i \leq k$, the following equation holds:

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

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 speed up that can be expected, as there is less data to be exchanged and thus less input for the global aggregation. Obviously, if an aggregate function is partitionable, the local aggregate function can be executed for all partitions S_i parallelly, while the global aggregation has to be processed sequentially.

If an aggregate function is used in combination with grouping, the optimizer can also decide to process several groups parallelly. In this case grouping can be done with the algorithms described in [39]. The algorithms discussed there can be applied orthogonally to our approach. Of course, if enough parallelism is possible by processing different groups parallelly, the optimizer might decide that no further parallel processing of the aggregate function is needed.

One disadvantage of this 2 step approach to parallel aggregation is that we are not always able to apply one

aggregate function to both sequential and parallel processing. Therefore the developer might have to implement and register six additional functions (Init, Iter, and Final functions for local aggregation and the same for global aggregation) to enable parallel as well as sequential processing of a UDAF. However, if one does not need maximum efficiency for sequential evaluation, one can simply use the local and global function for sequential execution, too. This, however, will incur at least the overhead for the invocation of an additional function. On the other hand, the additional work for the developer will pay off with all applications that are profiting from the increased potential for parallelism. Besides that, there seems to be no solution that results in less work for the developer.

4.2 Partitioning Classes and Partitionable Functions

One prerequisite for data parallelism is that one has to find a suitable partitioning of the data. This means that the partitioning must allow a semantically correct parallel processing of the function. In order to ease the specification of all partitionings that are allowed for the correct parallel processing of a UDFs, we describe a taxonomy of the functions that can be used for partitioning.

All 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 in some cases we will allow 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:

- ANY: the class of all partitioning functions. Round-robin and random functions are examples that belong to no other class. All partitioning functions that are not based on attribute values belong only to this class.
- 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.
- RANGE (column name [, N]): the class of partitioning functions that map rows, whose values of the specified column belong to a certain range, with the same multi-set of the result. Obviously there must exist a total order on the data type of the column. 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. 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 which are smaller than the values of a certain range have to be replicated into the resulting 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. The number of elements that belong to a certain range should be much greater than the value N. This class of partitioning functions is especially useful for *scalar* functions that require a sorted input, for example scalar functions that compute moving averages (see Section 4.5).

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).

Based on these definitions we can now define the classes of partitionable aggregate and scalar functions. These classes describe the set of UDFs that can be processed parallelly with the usual execution schemes for data parallelism (cf. Figures 2 and 3) and a particular class of partitioning functions.

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, $1 \leq i \leq k$, the following equation holds:

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

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, $1 \leq i \leq k$, the following equation holds:

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

The schemes in Figure 4 and Figure 5 show how partitionable functions can be processed parallelly. All k partitions can be processed parallelly. The actual degree of parallelism (i.e. mainly the parameter k) has to be chosen by the optimizer as usual. Please note, that for the scheme in Figure 4, there is not always a need to combine the local results. Hence, the optional combination step (computing $f(S) = \cup_{1 \leq i \leq k} f_l(S_i)$) is left out. In order to enable the DBMS to process a UDF parallelly the developer must specify the allowed partitioning class when the function is registered (cf. Section 4.4).

A scalar function f that is partitionable for class C using the associated function f_l can be evaluated parallelly 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 parallelly.

Figure 4. Parallel processing scheme for partitionable scalar functions

We have introduced some extensibility to the traditional parallel execution schemes by parameterizing the partitioning step by means of the partitioning function. In

An aggregate function f that is partitionable for class C using the two associated functions f_l and f_g can be evaluated parallelly using the following scheme, given an 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 $I_i := f_l(S_i)$ for $1 \le i \le k$ for all S_i parallelly.
   Send the intermediate results I_i to a single node for processing of step 3.
3. Compute $f(S) := f_g(\cup_{1 \le i \le k} \{I_i\})$;
   f_g can be applied to the I_i in arbitrary order.

Figure 5. Parallel processing scheme for partitionable aggregate functions

addition, we have defined classes of partitions to allow the optimizer more flexibility w.r.t. to the choice of the partitioning function. 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 specifies a single partitioning function for each UDF, in almost all cases a repartitioning step will be needed to process a UDF parallelly. Vice versa, if a single partitioning function satisfies all of the partitioning classes of a given set of UDFs, then repartitioning can be avoided.

Because UDFs can have arbitrary semantics, we believe that it is not possible to define a fixed set of partitioning functions that allows to apply data parallelism to *all* UDFs. If a given UDF is partitionable using some partitioning function p, but none of the partitioning classes defined above, the developer should be enabled to specify that this function p must be used. Using a special partitioning function should be avoided in general, since all data has to be repartitioned before such a UDF can be processed.

We want to remark here that implementing RANGE partitioning is a bit complicated, since a user-defined sorting order and partial replication have to be supported. One difficulty is for example 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 operator has to support user-defined sorting orders anyway. The definition of ranges and partial replication can be supported, if information about the data is collected during the sorting process.

In addition to that extension, the operator that invokes UDSFs has to be extended. 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 the moving average over five values is computed, 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. Though this extension is conceptually simple, it may be difficult to add it to an existing execution system.

4.3 Parallel Sorting as a Preprocessing Step for UDAFs

Some user-defined aggregate functions can be easily implemented, if their input is sorted according to a specified order. In this case the sort operation can be executed parallelly. Of course, this is especially interesting for UDAFs that are not partitionable.

Sorting as a preprocessing step for UDAFs can be introduced by using an additional parameter in the `CREATE FUNCTION` statement (see Section 4.4 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 are often user-defined data types. In most cases such functions will be needed anyway, to support sorted query results, to build indexes (like generalized B-Trees [42] or GiSTs [19]) or for sort merge joins to efficiently evaluate predicates on user-defined data types, to quote some examples.

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. We call this feature *early termination.* The parallel processing scheme for aggregate functions with sorted input is shown in Figure 6.

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 parallelly.
2. Compute f(S) without parallelism
   (use early termination, if possible).

Figure 6. Parallel processing scheme for aggregate functions with sorted input

The optional sort requirements can be integrated into rule-based query optimization (see e.g. [13], [25], [16], [28]) simply by specifying the sorting order as a required physical property for the operator executing the UDF. Then a sort enforcer rule ([13], [17]) can guarantee this order requirement by putting a sort operation into the execution plan, if necessary. Informix' Illustra [22] 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.

4.4 Extended Syntax for Function Registration

In this Subsection, we present the syntax extensions for the statements that allow the registration of UDFs with support for the features introduced in the previous Subsections.

Figure 7 shows the extensions for the `CREATE FUNCTION` statement. We have marked our extensions by boldface. The ORDER BY clause can be used to specify a sorting order that is required for the input table, on which the function is executed. The input table can be sorted on multiple columns applying user-defined sort functions to define the sort order. Furthermore the developer must specify, if early termination is used. To enable parallel evaluation, the partitioning class has to be specified. In addition to ANY, EQUAL and RANGE partitioning, the developer can register a special (user-defined) partitioning function for a UDF.

CREATE FUNCTION <function-name> (<argument type list>)
RETURNS <data type name>
EXTERNAL NAME <external function name>
[ORDER BY {<argument name> [USING <sort function name>] [ASC | DESC] } [EARLY TERMINATION]]
[ALLOW PARALLEL WITH PARTITIONING CLASS (
ANY
| EQUAL (<argument name list>)
| RANGE {<argument name> [, <number>]
[USING <sort function name>] [ASC | DESC]}
| <partitioning function name>)]
LANGUAGE <language name>
...

Figure 7. Extensions to UDSF registration

Figure 8 shows the extensions for the `CREATE AGGREGATE` statement. It now includes the local and global function options that are needed to register the aggregate functions that have to be used for the parallel evaluation of the new aggregate function. Of course the various Init, Iter, and Final routines that are registered must be consistent w.r.t. their argument types. For example the sequential and the global Final function must have the same return types (but often will have different argument types).

As we mentioned already in Section 3.2 additional information about these functions should be supplied by the developer. In addition to the usual cost parameters information about the size of the results of the local aggregation function (perhaps depending on the cardinality of the input set, if the function returns a collection type)

CREATE AGGREGATE <function-name>
(
<Init, Iter, and Final function definition>
[LOCAL <Init, Iter, and Final function definition>]
[GLOBAL <Init, Iter, and Final function definition>]
)

Figure 8. Extensions to UDAF registration

would be desirable.

4.5 Example Applications and Discussion

In this Subsection we present some example applications to illustrate the benefits of the introduced techniques.

4.5.1 Application to the UDAF most frequent

First, we will demonstrate how parallel execution can be enabled for the most frequent aggregate function.

How can we use the 2 step processing scheme to process the most frequent function parallelly? A straightforward approach could be to compute the most frequent value for each partition parallelly using the local aggregate function. This implies that the local aggregate function returns the most frequent value together with the number of its occurrences (i.e. the return type of the local function is a row type or a special user-defined type). Then the overall most frequent value is computed by the global function. Obviously this scheme is only correct if EQUAL is specified as the partitioning class for the local aggregation function. If ANY would be used as partitioning class, the local aggregate function would have to return *all distinct* values together with the number of their occurrences for each partition. Thus the local aggregation step would not be useful.

One difficulty of this approach is to implement the local aggregation function, since it must temporarily store *all* distinct values together with a counter. It is difficult to implement this efficiently in a user-defined function, since the function must be able to store an arbitrarily large data set. By contrast, the local aggregation can be done much easier if the developer uses sorting as a preprocessing step. The function must then only store two values and two counters: one for the most frequent value seen so far and one for the last value seen. This approach is much more practical. Based on the syntax from Section 4.4 we show the registration of the Iter function for the local aggregation ('$i' denotes the argument at position i in the parameter list of the function).

```
CREATE FUNCTION
        ITER_MF_LOCAL(POINTER, INTEGER)
RETURNS POINTER;
EXTERNAL NAME 'libfuncs!mf_iter_local'
ORDER BY $2 ASC
ALLOW PARALLEL WITH PARTITIONING CLASS
        EQUAL $2
LANGUAGE C ...;
```

Instead of using EQUAL and the ORDER BY clause, one could have also used RANGE as partitioning class. But this would have two disadvantages: first, all data must be sorted before the partitioning. With EQUAL as partitioning class the data is first partitioned and then only the partitions are sorted as specified in the ORDER BY clause. Second, repartitioning would occur more often due to the more restrictive partitioning class.

4.5.2 Application to the UDSF running average

As an example of a UDSF with input context, we discuss the running average function. This functions computes for each input value the average of the N values seen last. This means that the input context of the function is a 'window' of size N. Thus the running average function is partitionable of class RANGE with parameter N. Obviously the running average function computes many aggregates with a single scan over the input table. This is a typical example of a UDSF with an input context. Other functions of that kind are for example available in Red Brick Systems' Intelligent SQL [37].

4.5.3 Application to the UDAF Median

As an example of a function that seems to be not partitionable 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 that finds 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 9. Of course one would prefer a query using a UDAF `Median` as shown in Figure 10.

```
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 9. Computing the median in relational SQL

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

Figure 10. Computing the median in object-relational SQL

The object-relational 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 complex query in Figure 9 as we show in the following. The `Median` function is called with two parameters (cf. Figure 7): the first parameter is an element of the already sorted input set, the second parameter is constant and gives the cardinality of the input set[1]. 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 for 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. This implementation has an asymptotic complexity of O(N logN) due to sorting, while the computation with the SQL query from Figure 9 has one of $O(N^2)$. In case of the `Median` function using the early termination option would save roughly half of the calls to the function.

[1] Some systems do not support nesting of aggregate functions. In this case one could e.g. use a subquery in the FROM clause to compute the cardinality.

4.6 Summary

Table 1 shows the different kinds of contexts that can occur for UDFs and the implications for parallel execution with respect to data parallelism for aggregate and scalar functions. As can be seen from Table 1, our techniques support data parallelism with respect to many, but not all UDFs with input context. Additional techniques might emerge in the future. Please note that UDFs with external context are beyond the scope of this paper (c.f. subsection 3.2).

context	UDSF	UDAF
none	PARTITIONABLE WITH CLASS ANY	NOT REASONABLE
input	PARTITIONABLE WITH SOME CLASS *OR* NOT PARTITIONABLE	PARTITIONABLE WITH LOCAL & GLOBAL AGGREGATION AND SOME CLASS *OR* PARALLEL SORTING (& EARLY TERMINATION) *OR* NOT PARTITIONABLE
external	NOT TREATED HERE	NOT TREATED HERE

Table 1: UDFs and support for their parallel execution

5. RELATED WORK

User-Defined Functions (UDFs) have attracted increasing interest of researchers as well as industry in recent years (see e.g. [40], [42], [16], [27], [36], [32], [31], [1], [38], [18], [20], [6]). Despite this, most of the work discusses only the non-parallel execution of UDFs. We see our contribution as a generalization and extension of the existing work on the execution of user-defined functions using data-parallelism. In [32] pipeline parallelism for functions as well as intra-function parallelism are discussed. Intra-function parallelism allows a single function invocation to be processed parallelly. 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. Recently, IBM added the optional clause ALLOW PARALLEL or DISALLOW PARALLEL to the create function statement for UDSFs in DB2 UDB [21]. We view this as a first step of support for parallel execution of UDFs that is consistent with our more comprehensive framework. To the best of our knowledge there is no work on parallel processing of scalar user-defined functions with

an input context.

In [12] and [39] parallel processing of aggregate functions in RDBMS has been studied. The proposed concepts are applicable to built-in aggregation functions and consider also aggregation in combination with GROUP-BY operations and duplicate elimination. The proposed algorithms in [39] may be combined with our framework, if user-defined aggregate functions are used with GROUP-BY. It has been observed in [12] that different local and global functions are needed for parallel aggregation operations in RDBMS. In [36] the concept to process user-defined aggregate functions parallelly using two steps is proposed as a general technique, but neither are details nor more sophisticated processing techniques (like sorting as a preprocessing step, early termination or partitioning classes) presented. In [29] 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 [15] 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 pointed out that this classification is also useful for the parallel computation of user-defined aggregate functions. In the classification that is proposed in [15] an aggregate function f with a given input multi-set S and an *arbitrary* partition S_i of S is:

- *distributive* iff there is a function g such that $f(S) = g(\cup_{1 \leq i \leq k} f(S_i))$.
- *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.
- *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 with regard to 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 parallelly by our framework. Other holistic functions like e.g. the `Median` function can be efficiently evaluated in our approach, by using parallel sorting as a preprocessing step and early termination. Note that the application scenario in [15] is different to ours with regard to partitioning and parallel evaluation, because the sub-aggregates in data cubes must be computed for fixed partitions that are determined by semantically defined sub-cubes and not by the application of some partitioning function. 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.

In [44] 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 [44]. Thus for these partitionable functions also certain rewrite optimizations are possible that provide orthogonal measures to improve the performance. In [5] the class of group queries is identified for relational queries. This class is directly related to data partitioning. Our framework provides support for the concept of group queries in object-relational processing as well. Finally, we want to remark that [24] contains some additional examples for the application of our techniques.

6. SUMMARY AND FUTURE WORK

In this paper we have proposed a framework that allows parallel processing of a broad class of user-defined functions with input context in ORDBMS. This is an important step in removing a performance bottleneck parallelly object-relational query processing.

Since it was clear that a straightforward application of data parallelism is not possible, we had to devise more sophisticated parallelization techniques. The three key techniques that we have proposed here are the following: First, we have generalized the parallel execution scheme for aggregation in relational systems by means of local and global aggregations to allow its application to user-defined aggregations. Second, we have introduced some extensibility to the parallel execution schemes for scalar and aggregate functions by means of user-defined partitioning functions. We have defined classes of partitioning functions to make the specification of all allowed partitioning functions easier and to enable the optimizer to avoid data repartitioning as much as possible. Third, we have introduced parallel sorting as a preprocessing step for user-defined aggregate functions. This enables an easier implementation of UDFs and the use of parallelism in the preprocessing phase. Furthermore, we have defined new interfaces that allow the developer to use these techniques by providing the necessary information to the DBMS.

Some important remaining questions with respect to parallel object-relational query processing in general and especially UDFs are:

- Are there other classes of UDFs that do not comply with our methodology? As an example consider user-defined table functions [21] that can be used to encapsulate access to external data sources or external indexes [8].
- Though our framework supports parallel execution of user-defined predicates, we believe that substantial addi-

tional 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 an extension of our approach 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 [32] and collection types parallelly ([10], [3])?

Additional future work should be concerned with the extension of query optimization to our approach to parallel processing of UDFs.

7. ACKNOWLEDGEMENTS

We gratefully acknowledge the valuable comments of the anonymous referees, which have helped to enhance the presentation of this paper significantly as well as provided new insights.

8. REFERENCES

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

[2] Carey, M. J., Dewitt, D. J.: Of Objects and Databases: A Decade of Turmoil, VLDB 1996.

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

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

[5] Chatziantoniou, D., Ross, K. A.: Groupwise Processing of Relational Queries. VLDB 1997: 476-485.

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

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

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

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

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

[11] DeWitt, D. J., Carey, M., Naughton, J., Asgarian, M., Gehrke, J., Shah, D.: The BUCKY Object-Relational Benchmark, SIGMOD 1997: 135-146.

[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] Gray, J.: A Survey of Parallel Database Techniques and Systems, in: Tutorial handout at VLDB 1995.

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

[16] Haas, L. M., Chang, W., Lohman, G. M., McPherson, J., Wilms, P. F., Lapis, G., Lindsay, B. G., Pirahesh, H., Carey, M. J., Shekita, E. J.: Starburst Mid-Flight: As the Dust Clears. TKDE 2(1): 143-160 (1990).

[17] Haas, L. M., Kossmann, D., Wimmers, E. L., Yang, J.: Optimizing Queries Across Diverse Data Sources. VLDB 1997: 276-285.

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

[19] Hellerstein, J. M., Naughton, J. F., Pfeffer, A.: Generalized Search Trees for Database Systems. VLDB 1995: 562-573.

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

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

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

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

[24] Jaedicke, M., Mitschang, B.: A Framework for Parallel Processing of Aggregate and Scalar Functions in Object-Relational DBMS, TUM-I 9741, SFB-Bericht Nr. 342/25/97 A, September 1997. (http://www3.informatik.tu-muenchen.de/public/projekte/sfb342/publications.html).

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

[26] Mattos, N.: An Overview of the SQL3 Standard, Database Technology Institute, IBM Santa Teresa Lab, San Jose, California, July 1996

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

[28] McKenna, W. J., Burger, L., Hoang, C., Truong, M.: EROC: A Toolkit for Building NEATO Query Optimizers. VLDB 1996: 111-121.

[29] Ng, W., Levene, M.: OSQL: An Extension to SQL to Manipulate Ordered Relational Databases. IDEAS 1997: 358-367.

[30] Niblack, W., Barber, R., Equitz, W., Flickner, M., Glasman, E. H., Petkovic, D., Yanker, P., Faloutsos, C., Taubin, G.: The QBIC Project: Querying Images by

Content, Using Color, Texture, and Shape. Storage and Retrieval for Image and Video Databases (SPIE) 1993: 173-187.

[31] 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.

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

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

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

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

[36] 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.

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

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

[39] Shatdal, A., Naughton, J. F.: Adaptive Parallel Aggregation Algorithms. SIGMOD 1995: 104-114.

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

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

[42] Stonebraker, M., Moore, D.: Object-Relational DBMSs - The Next Great Wave, Morgan Kaufman Publishers, 1996.

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

[44] Yan, W. P., Larson, P.: Eager Aggregation and Lazy Aggregation. VLDB 1995: 345-357.