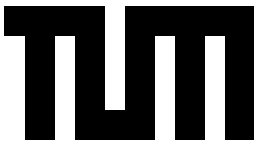

TECHNISCHE
UNIVERSITÄT
MÜNCHEN

INSTITUT FÜR INFORMATIK

Sonderforschungsbereich 342:
Methoden und Werkzeuge für die Nutzung
paralleler Rechnerarchitekturen

The Multi-Operator Method: Integrating Algorithms for the Efficient and Parallel Evaluation of User-Defined Predicates into ORDBMS

Michael Jaedicke, Bernhard Mitschang

TUM-I9910
SFB-Bericht Nr. 342/06/99 A
Mai 99

TUM–INFO–05-I9910-80/1.–FI

Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

©1999 SFB 342 Methoden und Werkzeuge für
die Nutzung paralleler Architekturen

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

The Multi-Operator Method: Integrating Algorithms for the Efficient and Parallel Evaluation of User-Defined Predicates into ORDBMS

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

Abstract

There has been a long record of research for efficient join algorithms in RDBMS, but user-defined join predicates in ORDBMS are typically evaluated using a restriction after forming the complete Cartesian product. While there has been some research on join algorithms for non-traditional data (e.g. spatial joins), today's ORDBMS offer developers no general mechanism that allows to implement user-defined join predicates in an efficient way. We propose the multi-operator method to achieve this goal and show that it is suitable to implement joins with complex user-defined predicates much more efficiently than today. Our approach fits well into the architectural framework of current ORDBMS. A significant further benefit is that the multi-operator method, in our view, can serve as an enabling technique for the parallel execution of complex user-defined functions.

1. Introduction

Object-relational DBMS (ORDBMS) are the next great wave in database management systems [40]. ORDBMS have been proposed for all applications that need both complex queries and complex data types. Many of these applications pose high requirements with respect to performance. Thus ORDBMS need to exploit parallel database technology. This has recently led to significant development efforts for parallel ORDBMS of some database vendors ([8], [32], [33], [34]).

Though this move to parallel ORDBMS stresses the demand for high performance, in our view, some techniques for efficient query processing are still missing. One obvious example is the support for join algorithms. There has been a lot of research on joins in RDBMS (see e.g. [12], [28] for a survey), but this topic has not been covered in depth for ORDBMS. The state of the art is essentially that user-defined join predicates in ORDBMS are evaluated by performing a nested-loops join that evaluates the user-defined predicate on the Cartesian product. A similar approach for handling traditional equality join predicates in RDBMS would make many queries slow. In ORDBMS the performance is deteriorated even further, since user-defined predicates are often extremely expensive to evaluate. Thus an evaluation of these predicates on the full Cartesian product (even if not materialized) results in prohibitively high costs and unacceptable performance.

In the context of special application areas, there has been some research on efficient join algorithms

for data types like spatial data (see e.g. [4], [35], [36], [42]). Unfortunately, some of these techniques cannot be integrated well into current ORDBMS. The reason is that the current implementation model for user-defined functions is too simple to allow for sophisticated implementations.

Our main contribution in this paper is to propose the multi-operator method as an implementation technique to solve these fundamental problems. The multi-operator method allows to integrate complex implementations of user-defined functions and predicates in ORDBMS with the best possible support for parallel evaluation. The basic idea of the multi-operator method is to use multiple operators instead of a single one which allows the decomposition of the user-defined function. This results into a simple, yet powerful technique. We believe that it offers the potential for immense performance gains for many object-relational applications. Performance measurements that we present later demonstrate this.

Our approach fits well into the overall architectural framework of current parallel ORDBMS. Thus we are convinced that this method is easy to use for developers and can be supported in ORDBMS without major difficulties. In fact, as we discuss later, there are other reasons for database vendors to support this kind of interface.

The rest of this paper is organized as follows. We give an introduction to the problem in Section 2 along with some terminology. Section 3 introduces and discusses the multi-operator method. We discuss a spatial join algorithm as an example application. Section 4 discusses the interface that is necessary to make the multi-operator method available in ORDBMS. Section 5 reports on performance measurements for our spatial join example and indicates significant performance gains. We cover the related work in Section 6 and conclude the paper with a summary in Section 7.

2. User-Defined Functions in Object-Relational Query Processing

We will now provide the basic concepts and definitions that 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 ([12], [11], [2], [29]) and object-relational query processing ([40], [6], [4], [35], [17], [40]). After an introduction to user-defined functions in Subsection 2.1, we discuss the problem that we address here in Subsection 2.2, and in Subsection 2.3 we present a running example.

2.1. User-Defined Functions and Predicates

Every RDBMS comes with a fixed set of built-in functions. These functions can be either scalar functions or aggregate functions (also called set or column functions). A *scalar function* can be used in SQL queries wherever an expression can be used. Typical scalar functions are arithmetic functions like + and * or `concat` for string concatenation. A scalar function is applied to the values of some columns of a row of an input table. 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. Typical aggregate functions are `MAX` and `SUM`.

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* and *user-defined aggregate functions*. A user-defined scalar function that returns a boolean

value is a *user-defined predicate* (**UDP**).

2.2. Performance Problems with Complex UDFs in Current ORDBMS

We claim that there is currently not enough support for the efficient implementation of complex UDFs in ORDBMS. We will demonstrate this in the following using a spatial join as an example of a complex UDP. Our example is the following spatial query, which selects all pairs of polygons from a single table that overlap with each other:

```
select      *
from        Polygon_Table a, Polygon_Table b
where       overlaps(a.p_geometry, b.p_geometry)
```

How is this query evaluated in a current commercial ORDBMS? Let us assume that no spatial indexes can be used to evaluate the join predicate `overlaps`. Because this is not a traditional join (like an equi join), no system will use a hash or a merge join. Hence, all ORDBMS will use a nested-loops join for evaluation, i.e. the UDP `overlaps` has to be evaluated on the full Cartesian product. Please note that in this scenario data parallelism can be applied to the nested-loops join by partitioning the outer input stream and replicating the inner input stream (at least, if the system supports data parallelism for UDFs [23]). In general the developer must be able to disallow such a parallel evaluation, since the function that implements the UDP might have a global state or side effects. In fact, most sophisticated algorithms have such a global state or side effects, since it is often necessary to hold temporary data.

Despite parallel evaluation the performance of this straightforward approach is poor. It is known that a much better performance can be achieved, if a filter and refine approach ([35], [36], [42], [4]) is taken. This approach uses two phases: the filter phase finds pairs of candidates for the result using a relatively inexpensive test. This test is only a rough filter in the sense that some false candidates can pass the test. All candidate pairs are therefore tested with the original join predicate (the overlaps predicate) in the refinement phase. It has been observed in the literature that this approach results in a much better performance, as the filtering reduces the input for the refinement phase significantly and the latter is the most expensive one. Typically the test in the filter phase is not evaluated using the exact geometries. Rather a simple approximation like the bounding box is used for the test. This reduces the CPU costs for the test and also reduces the storage requirements considerably. We will provide more details in the next Subsection.

2.3. The PBSM Algorithm as an Example of a Sophisticated UDP Implementation

We will now use the Partition Based Spatial-Merge Join algorithm (PBSM) proposed in [36] for a detailed example. We want to emphasize again that we are not aware of a practical way for developers of DBMS class libraries [4] to implement such a sophisticated algorithm in any ORDBMS. It is the goal of the multi-operator method to make this possible with full support for parallel evaluation.

The PBSM algorithm is based on a two-phase filter-and-refine-scheme and is processed sequentially, though the authors of [36] believe that it can be parallelized efficiently. We will first give an outline of

the algorithm. The input of the PBSM are two sets of spatial object geometries (like polygons). It is assumed that each spatial object can be identified via an OID. We will assume that the spatial join predicate is the `overlaps` function. We will number the steps of the algorithm using numbers in brackets like [1.1]. The first number denotes the phase, the second one the step within this phase.

The first phase is the filtering phase. It begins with creating key-pointer elements (step [1.1]), i.e. (MBR, OID) pairs, where MBR denotes the minimum bounding rectangle. These key-pointer elements are then written to temporary tables on disk (step [1.2]). Next a decision is made (step [1.3]): if the temporary relations fit into main memory, the algorithm proceeds with step [1.4], otherwise with step [1.5]. In step [1.4] the candidate pairs are computed using the MBRs and a filter algorithm. The filter algorithm is based on plane-sweeping techniques from computational geometry and is viewed as a kind of spatial-merge join in [36]. This set of candidate pairs forms the input for the refinement phase. In step [1.5] the data for the filter step is first partitioned. This begins with an estimation of the spatial universe (the MBR of all spatial input objects) using data from the system catalog. Step [1.6] decomposes this universe into P subparts. Given this partitioning of the space, the corresponding spatial data partitioning function is applied to the temporary tables that contain the key-pointer elements (step [1.7]). Finally, partitions with the same index are processed with the filter algorithm (step [1.8]). This produces the set of candidate OID pairs for the refinement step.

The second phase, refinement, proceeds as follows. First, duplicates in the set of candidate pairs are removed (step [2.1]). Such duplicates can occur, since the key-pointer elements are partially replicated during the partitioning (to deal with spatial objects overlapping several partitions). In step [2.2] the OID pairs are sorted on the OID of the first input (this will avoid random I/O). Then a subset of the base data of the first input is fetched that fits into main memory and the corresponding OIDs are swizzled to main memory (step [2.3]). Next the corresponding OIDs of the other input are sorted (step [2.4]). The tuples corresponding to these OIDs are then read sequentially from disk (step [2.5]) and finally the exact join predicate (i.e. `overlaps`) is evaluated (step [2.6]). The algorithm proceeds by returning to step [2.3], until all candidate pairs have been processed (step [2.7]).

Please note that the PBSM must manage intermediate results, retrieve objects based on their OID and partition data using a special partitioning scheme. *Currently there exists no technique that allows developers to integrate an algorithm with such features into an ORDBMS.* Note that the complexity of the PBSM algorithm is by no means an exception. As pointed out in [35] for the geo-spatial application domain, complex UDFs often consist of multiple steps.

3. The Multi-Operator Method as a New Technique to Implement Complex UDFs

Next we will describe our approach to overcome the performance problems discussed in the previous Section. We introduce the multi-operator method in Subsection 3.1 and discuss its principal benefits. Subsection 3.2 illustrates the method by studying a multi-operator implementation of the PBSM algorithm as an example.

3.1. The Multi-Operator Method and its Benefits

The basic idea of the multi-operator method is quite simple. Current ORDBMS allow the implementation of a UDF only by means of a function that is executed within a *single* operator (e.g. a restriction, projection or a nested-loops join) of a query execution plan. By contrast we will allow an implementation by means of *multiple* operators (i.e. by an operator graph, where the operators may invoke other UDFs).

This method can be viewed as an extension and generalization of the well-known implementation scheme used in relational engines for sort-merge joins. Here, instead of a single operator typically three operators are used to implement this join algorithm: one sort operator for each of the inputs and an operator for the merge join. There are at least three good reasons for this design. The first reason is that a sort operation is needed for other purposes (like producing a sorted output), too. Thus this operator can simply be *reused*. The second reason is that sometimes an input stream may already arrive sorted from previous operations - in this case no sort is needed and the superfluous sort operation can be eliminated. In general, *optimizations* of the query execution plan are enabled. The third reason is that using three operators instead of one allows to use more *parallelism*. For example, the two inputs can be sorted in parallel by means of the highly optimized system-provided parallel sort operation. All of these arguments - reuse, optimizability, and parallelism - are in our view applicable to multi-operator implementations of UDFs in general. This is not so much surprising, given that the operator concept is one of the basic abstractions and building blocks used in query execution. Therefore replacing a single complex operator by a set of simple ones offers the possibility for improved query execution plans due to the finer granularity that is introduced.

As we will show below in detail, a multi-operator implementation supports all kinds of parallel execution, i.e. intra-operator parallelism, pipelining and independent parallelism. Even if a traditional implementation by means of a single function is possible, it is often impossible to execute the operator containing this function with data parallelism due to the complexity of the UDF implementation (e.g. side effects, storage of temporary data). By contrast, a multi-operator implementation allows the developer to code the UDF in a set of operators, where each operator can possibly be executed in parallel and where the DBMS knows the dataflow between the operators. We believe that the multi-operator method provides the developer with the right method to provide the DBMS with an implementation that can be executed in parallel, since operators are a natural granule for parallel execution in relational database systems ([11], [12]).

The multi-operator method can achieve especially high performance gains when it is applied to UDPs that can serve as join predicates as described in Subsection 2.2. In this case, a multi-operator implementation will allow to replace the standard execution plan consisting of a Cartesian product and a restriction with the UDP by a multi-operator implementation of that UDP that is a user-defined join algorithm that exploits multiple operators and several UDFs. It is easy to see that this will enhance the performance of many object-relational queries, when one considers the importance of joins in relational systems: without the various highly optimized join algorithms that avoid Cartesian products for the most commonly

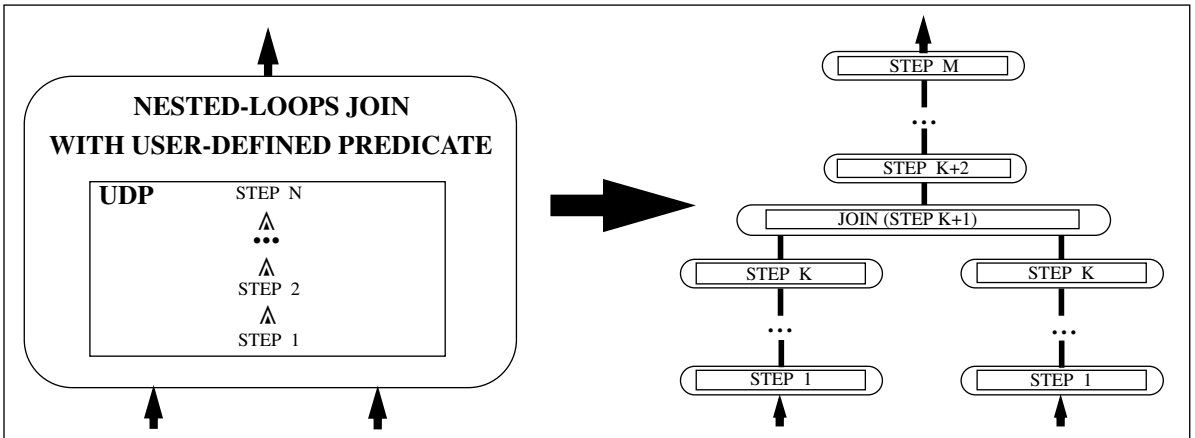

Figure 1: Multi-operator implementation of a UDP with implicit join

used join predicates, relational systems would have been hardly successful in the marketplace.

Figure 1 shows a schematic view of a multi-operator implementation for a UDP. Without special support a UDP is evaluated by means of a nested-loops join on the complete Cartesian product as shown on the left side of Figure 1. A complex UDF often executes a number of steps, here N. The multi-operator implementation allows to perform a number of processing steps (K steps in Figure 1) before the two input relations are actually joined. Thus a part of the UDP can be pushed down. Please note, that now no Cartesian product is needed. Instead, a specialized join algorithm (Step K + 1) can be used. After the join some postprocessing is done. In general, the preprocessing steps will have often the goal to map user-defined data types to approximations (sometimes also called features) that are represented by built-in data types. These approximations can then be processed with the traditional built-in operators to perform a filter step. The postprocessing steps then do the refinement based on the results of the filter step.

Note that the number of steps in the single-operator (N) and the multi-operator implementation (M) may differ as already mentioned before. In general the design of both implementations may vary significantly. If the developer does a careful design, parallelism can be exploited to a full extent in the multi-operator implementation.

3.2. A Multi-Operator Implementation of the PBSM Algorithm

To provide a detailed example application of the multi-operator method, we will now discuss, how the PBSM algorithm from Subsection 2.3 can be redesigned and mapped to a multi-operator implementation (no details concerning the integration of PBSM into Paradise [35] are given in [36], but it seems to be implemented by a new database operator). Of course, one goal of such a redesign should be to allow as much parallelism as possible for the multi-operator implementation. Figure 2 shows on the right side a multi-operator implementation of a simplified version of the PBSM algorithm. In the first step, the key-pointer elements are generated for both inputs. These key-pointer elements are then partitioned. It is necessary to have a common spatial partitioning for both input streams to be able to compute the join by joining only corresponding partitions. The next operator joins the partitions and generates the

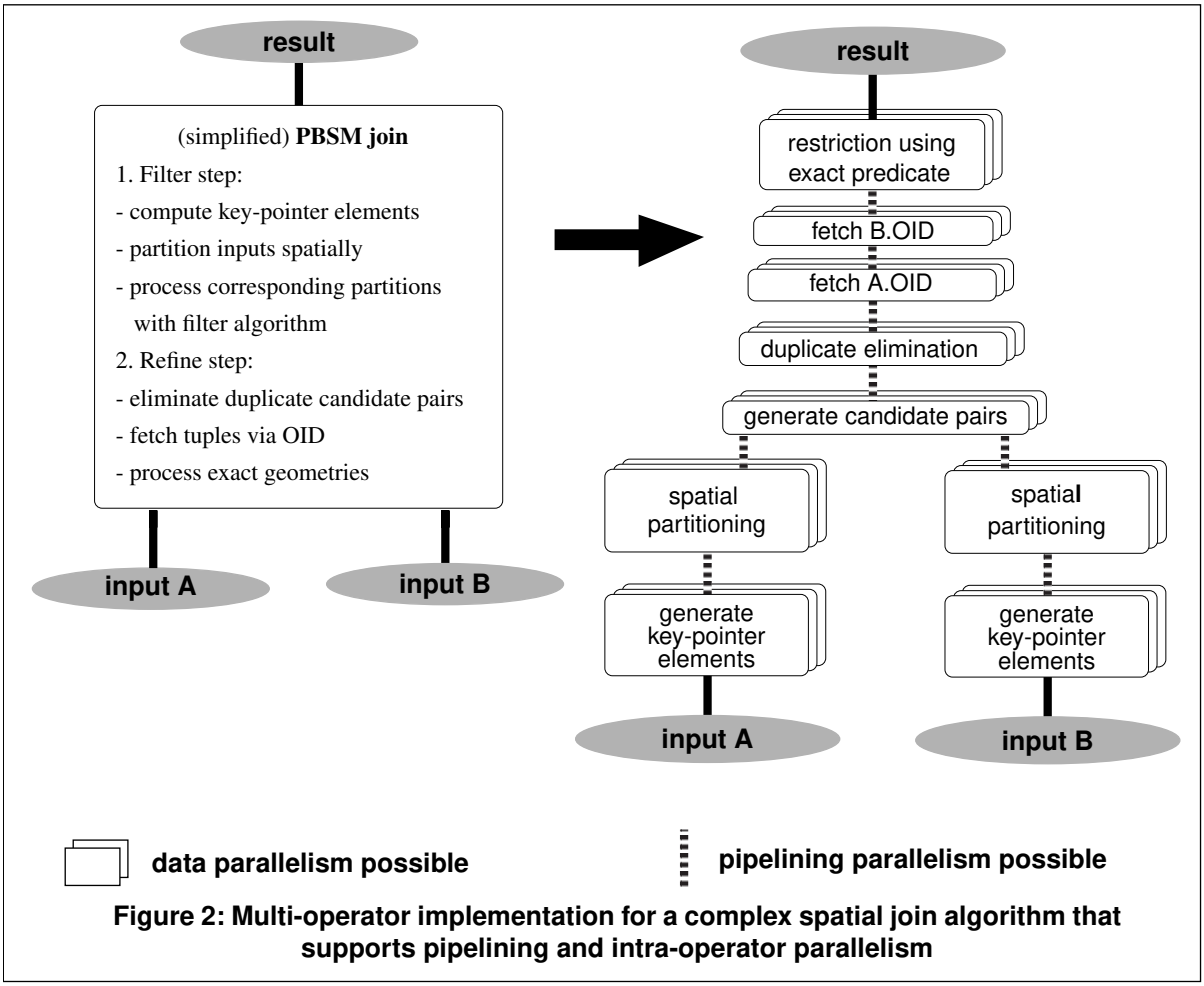

Figure 2: Multi-operator implementation for a complex spatial join algorithm that supports pipelining and intra-operator parallelism

candidate pairs. This corresponds to step [1.8] of PBSM. The second phase, refinement, starts with the duplicate elimination. Then the next two operators fetch the exact geometries from disk using the OIDs. Finally, the spatial join predicate is evaluated on the exact geometries for each candidate pair.

Several remarks w.r.t. the multi-operator implementation are in order. Our redesign did really simplify the PBSM algorithm: there is no decision on the processing strategy based on the amount of temporary data (cf. step [1.3] above). In a parallel processing scenario this is desirable, since it avoids a blocking of the evaluation, until the decision is made. Furthermore, the refinement step was simplified substantially, as there are no efforts to avoid random I/O. Though this will lead to higher resource requirements for disk I/O, we believe it is much harder to avoid random I/O, if parallel evaluation is used. There are also two major advantages of the overall design. First, pipelining parallelism can be applied between all operators. Second, data parallelism can be used for all operators. Especially, the join and the very expensive predicate in the refinement phase can be evaluated in parallel (see Figure 2).

When one compares this implementation with the original PBSM as described above, it is obvious that all control structures (like conditional statements and loops, cf. steps [1.3] and [2.7] in Subsection 2.3) have been eliminated. *As a direct benefit of this redesign, query parallelization as known from traditional relational DBMS is enabled and in addition the processing model of relational execution*

engines is matched perfectly. Thus it allows to implement efficient join algorithms for UDPs that can be evaluated in parallel by existing object-relational execution engines.

4. Supporting the Multi-Operator Method in ORDBMS

The multi-operator method fits well into current system architectures. We want to point out here that its implementation in a commercial ORDBMS requires *no changes of the core data base system*. It is only necessary to add a new interface to the current APIs (like embedded SQL, ODBC and JDBC) that allows the execution of a query execution plan that is specified by the developer. In the following we will argue why we believe that such an interface is useful for current DBMSs anyway.

We believe that there are several reasons why an interface that allows the specification and execution of query execution plans is needed. We want to make clear that such an interface will only be used in addition to SQL, not as a replacement. In our opinion this interface will typically be used by sophisticated developers of third party software vendors that produce packaged applications like enterprise resource planning software, geographic information systems, CAD software etc.

The first reason is that very often developers have the problem that they know an execution plan that is better than the plan that is generated by the optimizer. There are many reasons for this situation like for example erroneous selectivity estimation. Currently there is only the possibility to view the plan that the optimizer has generated. If this plan is not the desired one, one can then try to manually transform the corresponding SQL statement until one reaches a more satisfying result. This process can be both time consuming and disappointing. There is a further shortcoming of this approach in practice. When vendors upgrade their DBMS they often improve their optimizer somehow. While such improvements may be beneficial for most queries they can degrade the performance of some others - maybe just the important ones. Therefore developers face the problem that the optimizer might change the plan of the 'tuned' SQL query in the next release. Moreover the choice of the concrete execution plan might also depend on database statistics and the settings of the various tuning knobs of the DBMS both of which may vary from customer to customer. This makes it pretty hard to guarantee a good performance for a specific query and to provide customer support. Obviously all these problems can be avoided if the execution plans are specified by the developer. On the other hand all these particular situations could be taken care of by a comprehensive and sophisticated optimizer. However optimizer technology is not yet there. Hence it can be seen as a conceivable step by some vendors to add support for this feature (cf. [1]; G. Lohman, personal communication).

The second reason for supporting query specification at the plan level is that the full power of the query execution system cannot be unlocked by SQL. For example, it is well-known that in SQL sorting can only be used in the outermost query block. This is sometimes too restrictive in practice [23]. When developers can specify query execution plans directly they can go beyond these limits.

We believe that these points themselves justify the extension of the current APIs. The multi-operator method adds just another reason, why such an interface is highly desirable. However, there is also a drawback of this approach, if we add no further enhancements: without any query optimization, i.e. when the query execution plans are executed exactly as specified by the developer, the plan is not auto-

matically adapted to changes of the database like a drastically increased table size or changed value distributions. An improvement of this situation is to allow the DBMS the optimization of *some, but not all properties* of a manually specified execution plan. For example one can allow the optimizer to remove superfluous sort operators, or to optimize the join order, or to optimize the implementation methods of the plan (e.g. switch from merge join to hash join), or to choose a different access path (using a table scan instead of an index scan). In a rule-based optimizer like Cascades [13] this can be supported by grouping the rule set into subsets. Then only some subsets of the whole rule set (for example only the rules that eliminate superfluous sort operators) are activated when a manually crafted plan is optimized. In this case, the developer must be enabled to specify which properties of the plan can be optimized. Such specifications can be viewed as optimizer hints (see also [1]).

One could object that it is difficult for a developer to come up with a plan for a complex query like one that involves many tables, many predicates and subqueries. However, the developer can always use a query execution plan that is generated by the optimizer as a starting point. This plan can then be modified by the developer. For example, this allows to start with a join order that the optimizer considers as good for a given database.

There are different possibilities how query execution plans can be specified. One possibility is to store plans in tables (query explain tools of some vendors do this already [18]) and then allow developers to update these plans and execute the modified ones. Graphical tools that visualize operator graphs and allow their interactive manipulation are certainly desirable. Although interfaces for query execution plans are not the focus of this paper, we describe the interface of our prototype that is based on a textual description of operator graphs in Subsection 5.2.

5. Performance Evaluation

In this Section we will present initial performance measurements that indicate a significant performance increase for the evaluation of complex UDFs according to the multi-operator method. Our example is a query with a spatial join, similar to Subsections 2.2 and 3.2.

For our measurements we used the following environment:

- Hardware:
  We performed our experiments using a SUN Ultra-2 workstation (model 2200) with 2 Ultra SPARC-1 processors (200 MHz each), 1MB external cache, 512 MB main memory, and 2 SUN 4 GB hard disks.

- Software:
  We conducted our initial performance measurements using our parallel ORDBMS prototype MIDAS. MIDAS supports full intra-query parallelism on symmetric multi-processors (SMPs) and networks of workstations and SMPs. Please refer to [31] and [3] for details.
  To implement the different query execution plans for the spatial join we used an interface that allows to describe query execution plans and that is described in Subsection 5.2. In addition we defined and implemented some UDFs (using an interface similar to that of commercial ORDBMS).

5.1. Experimental Scenario

We will demonstrate the performance of the multi-operator method for the example query from Subsection 2.2 that selects all pairs of overlapping polygons from a table with polygon geometries. We generated random data for our experiments due to two reasons. First, this eased control over our experimental setting and second, we only wanted to demonstrate the benefits of the multi-operator method. Especially, we were not focused on designing new algorithms for spatial joins. Thus using real application data seemed not to be critical to our experiments.

We used 3 tables PG1, PG2, and PG3 that contained 1 000, 10 000 and 100 000 regular N-corners as polygons. The polygons were generated with 3 random parameters: N (the number of corner points of the polygon), the radius (distance of the corners to the center) and the location of the center in the 2 dimensional plane. The polygons of table PG2 were placed in a rectangle (our spatial universe) bounded by the points (0, 0) and (100 000, 100 000). For the table PG1 we reduced the area of the spatial universe by a factor of 10, for table PG3 we enlarged the area by a factor 10. Therefore, in the average the number of polygons in an area of constant size is about the same for all tables. This results in a linear growth in the number of overlapping polygon pairs in the three tables. The number of points per polygon was randomly chosen between 3 and 98 with an average of about 50. The radius was chosen as $20 + N + \lfloor r * N \rfloor$; the value r is a random number between 0 and 1. This means that the more points a polygon had, the larger its area. Table 1 provides some statistics about the three tables. As one can see, the number of polygon pairs that actually do overlap increases roughly linearly with the cardinality of the table.

We stored the polygon data as a list of points in a VARCHAR data type and defined 3 tables PG1, PG2, and PG3 to store the polygons together with an integer value as primary key:

```
CREATE TABLE PG1(  INTEGER  id PRIMARY KEY,
                   VARCHAR  p_geometry)
```

The tables were stored on a single disk. We then implemented the UDF `overlaps(polygon1, polygon2)` that tests, whether the two argument polygons overlap. The function returns an integer value of 1, if the two polygons overlap geometrically and 0 otherwise[1]. We used the C++-library LEDA [30] for a rapid implementation of the geometric functionality.

To avoid including the time to display the result in the measurements, we simply counted the number of result tuples. Given these explanations, our test query was simply the following, with Polygon_Table being PG1, PG2, or PG3:

Test Query:

```
select      count(*)
from        Polygon_Table as a, Polygon_Table as b
where       overlaps(a.p_geometry, b.p_geometry) = 1
```

The first implementation of the UDF `overlaps` is straightforward: simply test whether the polygons

1.A Boolean return type would be more appropriate, but MIDAS currently does not support this.

Table 1: Statistics of the test data tables

	table PG1	table PG2	table PG3
cardinality of table	1 000	10 000	100 000
disk space (KB) [clustered in B-tree]	736	6 464	71 584
avg(# corner points / polygon):	48.35	49.97	50.01
avg(area(polygon))	31 276.7	32 689.8	32 754.3
# overlapping bounding boxes	1 188	12 144	121 460
# overlapping polygons	1 140	11 550	115 336

overlap using the exact geometry (the implementation uses a (slightly modified) plane-sweeping algorithm from computational geometry provided by the LEDA library). In the following we will call this function `exact_overlaps`. Now we will explain how we tried to speed up the evaluation of the `overlaps` predicate. The first step was to use a simple filter and refine scheme to improve the efficiency of the implementation. The filtering step uses bounding boxes as approximations of the polygons. Bounding boxes were represented by their lower left and upper right corner coordinates and were also represented as strings. We implemented this in the following function:

- `filter_overlaps(polygon1, polygon2)`:
  Generate and overlap bounding boxes for the input polygons first, then overlap their exact geometries, if necessary.

While `filter_overlaps` improves the performance of a single-operator implementation, it does not support a multi-operator implementation based on the PBSM algorithm as described in Subsection 3.2. Thus we implemented three additional UDFs to provide suitable building blocks for the multi-operator method:

- `bbox(polygon)`:
  Create a bounding box for a given polygon.
- `bbox_overlaps(bbox1, bbox2)`:
  Test whether two bounding boxes overlap.
- `bbox_partition(bbox)`:
  Compute all buckets to which a given bounding box has to be copied.

The UDF `bbox_overlaps` does a relatively simple geometric test and is very inexpensive compared to the `exact_overlaps` function for polygons. The partitioning function `bbox_partition` divides the spatial universe in B equally sized rectangular regions that we call buckets. The function then computes all buckets, which a given bounding box overlaps. Since we divided both dimensions T times B is equal to T^2. The actual implementation has additional parameters not shown in the prototype of the function that define the data partitioning (e.g. the value T).

Now we are ready to describe the four implementations of the UDP `overlaps` that we have evaluated[1]:

- **Plan A (naive implementation):**
  Execute the UDF `exact_overlaps` on the Cartesian product $(a \times b)$.
- **Plan B (filter and refine within a single UDF):**
  Execute the UDF `filter_overlaps` on the Cartesian product $(a \times b)$.
- **Plan C (nested-loops join only with bounding boxes):**
  Generate the bounding boxes with the UDF `bbox` for both relations `a` and `b` and store the bounding boxes of the inner relation `b` temporarily together with the primary key column id. Then build the Cartesian product of the result. Evaluate the UDF `bbox_overlaps` next. For all candidate pairs retrieve the polygon data (via a B-tree) using a join on the `id` value and finally evaluate the UDF `exact_overlaps`. Altogether plan C employs 4 UDFs in the plan: `bbox` (2x), `bbox_overlaps` and `exact_overlaps`.
- **Plan D (spatial partitioning and merge join):**
  Plan D extends plan C by using spatial partitioning for the bounding boxes by means of the function `bbox_partition`. The bucket number is appended to each temporary tuple. This allows to join the tuples that belong to corresponding buckets using a merge join on the bucket number. Thus the function `bbox_overlaps` is evaluated only on the set of all pairs of bounding boxes with the same bucket number (and not on the complete Cartesian product like in plan C). Next the elimination of duplicate candidate pairs is done and finally the UDP `exact_overlaps` is evaluated on the exact geometries. Altogether plan C employs 6 UDFs in the plan: `bbox` (2x), `bbox_partition` (2x), `bbox_overlaps` and `exact_overlaps`.

Please note that plans A and B are still traditional single-operator implementations, whereas plans C and D are multi-operator implementations. Furthermore, the quality of the UDP implementation steadily increases from the straightforward solution of plan A to the sophisticated solution of plan D, which is very similar to the redesigned PBSM algorithm (cf. Subsection 3.2). These four plans were specified using a textual notation that we describe in the following Subsection.

5.2. Textual Specification of Query Execution Plans in MIDAS

In MIDAS query execution plans can be specified in a text notation for internal test and debugging purposes. However, one can also execute such plans from an application via the call level interface.

Figure 3 shows the textual description of plan C. Figure 4 presents the operator graph of this plan with well-known terms for the operators. As can be seen in Figure 3, each node of the operator graph is limited by parentheses, has a unique label and contains the name of its operation and additional parameters. An operator is either a bulk operator (that operates on a table like a restriction) or an item operator (that operates on a tuple and serves to specify the functionality of a bulk operator - e.g. the predicate of a restriction is specified by means of item operators). The sons of a node are nested within the description of the node itself, which results in a LISP-like notation.

Next, we describe some of the operators that are shown in Figure 3. The SEL node represents the top operator of SELECT queries. The GROUP operator does grouping and aggregation (the aggregate functions are given as parameters of the node) and the TIMES operator builds a Cartesian product. The

1.We want to remark here that our prototype currently does not support real nested-loops joins, but is only able to perform a restriction after forming the Cartesian product. But since the execution is demand-driven and tuple-at-a-time, the intermediate Cartesian product is not materialized. In addition, MIDAS does not support OIDs. For our tests, we replaced OIDs simply by the primary key `id`.

RESTR operator performs a restriction, the REL operator does a relation scan (the table name is specified as a parameter). The EQ node specifies an equality predicate, the CONST node generates a constant value of a specified data type and the ATTR node fetches an attribute value. One can refer to the attributes of the input tuple of a parent node by means of the label of this node and the position of the attribute within the tuple. The FUNC node allows to evaluate a (user-defined) function whose name is given as parameter. Projections are done by the PROJ node that has a BUILD node as right son. The sons of the BUILD node specify the attributes that appear in the output tuples of the PROJ node. A REL node can also be used to describe an index scan, if it has an IVMK node as son. The latter serves to specify the key interval that must be accessed. The CORR node materializes an intermediate table and can also be used to evaluate correlated subqueries.

```
(N1:sel   { no_update }
   (N2:group { count[*] }
      (N3:restr
         (N4:times
            (N5:times
               (N6:proj
                  (N7:restr
                     (N8:times
                        (N9:proj
                           (N10:rel   { pg1 } )
                           (N11:build
                              (N12:attr  { N9[1] } )
                              (N13:func  { 'bbox' }
                                 (N14:attr  { N9[2] } )   )))
                        (N15:corr
                           (N16:proj
                              (N17:rel   { pg1 } )
                              (N18:build
                                 (N19:attr  { N16[1] } )
                                 (N20:func  { 'bbox' }
                                    (N21:attr  { N16[2] } ) ))))))
                     (N22:eq
                        (N23:func  { 'bbox_overlaps' }
                           (N24:attr  { N7[2] } )
                           (N25:attr  { N7[4] } ) )
                        (N26:const { 1 integer } ) ))
                  (N27:build
                     (N28:attr  { N6[1] } )
                     (N29:attr  { N6[3] } ) ))
               (N30:proj
                  (N31:rel   { pg1 }
                     (N32:ivmk  { eq }
                        (N33:attr  { N5[1] } ) ))
                  (N34:build
                     (N35:attr  { N30[2] } ) )))
            (N36:proj
               (N37:rel   { pg1 }
                  (N38:ivmk  { eq }
                     (N39:attr  { N4[2] } ) ))
               (N40:build
                  (N41:attr  { N36[2] } ) )))
         (N42:eq
            (N43:func  { 'exact_overlaps' }
               (N44:attr  { N3[3] } )
               (N45:attr  { N3[4] } ) )
            (N46:const { 1 integer } ) )))
)
```

Figure 3: Textual specification of plan C in MIDAS

Syntactic and semantic checks must be done when the description of an execution plan is parsed and transformed into the internal representation of the DBMS. In MIDAS a part of the routines that perform the semantic checks for SQL statements are simply reused for this task. The text notation specifies the structure of the operator tree, the implementation methods for the operators (e.g. the join methods) and provides additional parameters for the operators (for example, the name of a UDSF). Additional parameters that are needed for the execution like the amount of main memory that should be used for sorting are set by the compiler based on cost estimations or default values. The system can also convert the execution plans of compiled SQL statements into this textual format to present them to developers.

Without optimizer hints an execution plan is executed 'as is' by the DBMS. However, the developer can also specify that the following optimization phases should be applied to the plan: join ordering, transformation of subqueries to joins (if possible), sort elimination, optimization of projections, optimization of local restrictions, index access, use of temporary tables, and

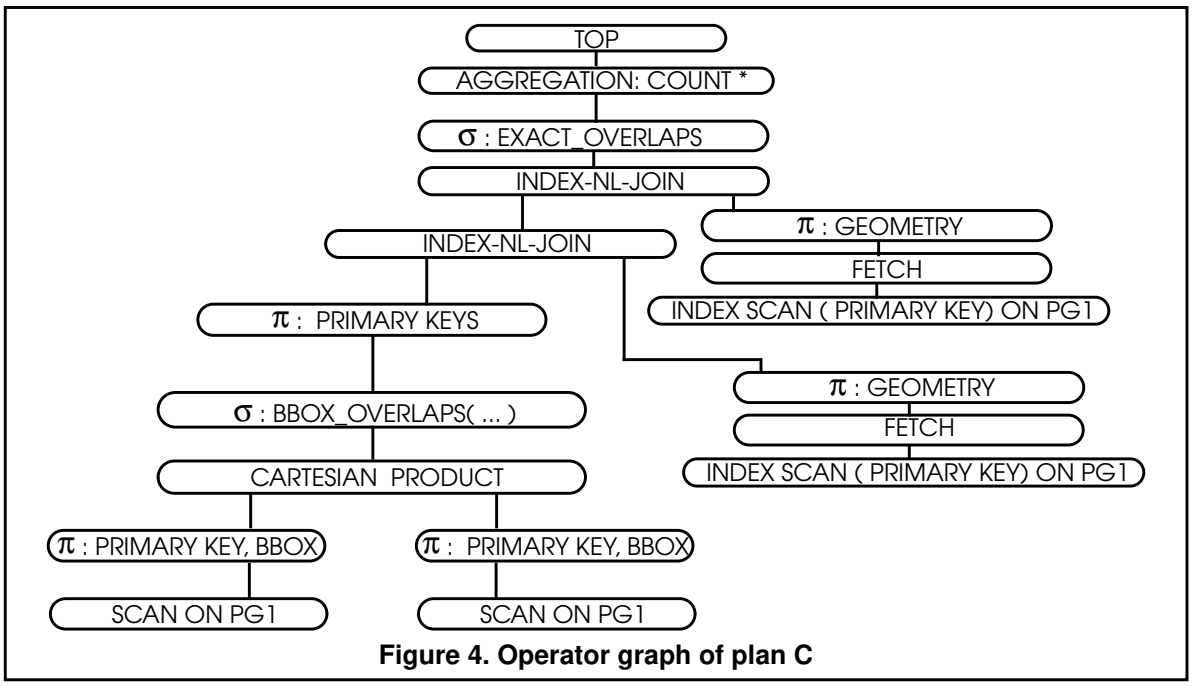

Figure 4. Operator graph of plan C

optimization of data modification commands. These options add much flexibility for developers. We want to mention here that this interface was already available in TransBase, the commercial system that served as base for the MIDAS development [41].

The parallelization approach of MIDAS [31] is operator-based. This means that data parallelism, independent parallelism, and pipeline parallelism can be introduced by integrating specific SEND and RECEIVE operators into the plan that manage the data partitioning, i.e. split and combine data streams. The parallelization can be done by the parallelizer of the DBMS itself (which is based on Cascades [13]) or the developer can specify this manually. The latter is done by inserting SEND and RECEIVE nodes into the textual description of the sequential execution plan. The other operators do not have to be modified. For automatic parallelization the UDFs that are used in the multi-operator implementation have to be registered with meta data that describes how they can be parallelized (see [23] for details).

5.3. Performance Results

We present now some initial performance results. With respect to the absolute execution times, please note that we used our prototype database system and did not fine tune our UDF implementations. But the relative performance measures should give a first impression of the performance gains.

In Table 2 we present the sequential execution times in seconds for all plans. The last row shows the time that is needed to do the exact geometry processing for the candidate pairs (refinement). Some execution times are not available, since we did not evaluate plans with execution times higher than 100 000 seconds (nearly 28 hours). Several conclusions can be drawn from Table 2: First, there are significant performance increases for plans B, C, and D. Processing queries with expensive UDFs on large tables in acceptable times obviously requires sophisticated implementations. Second, the multi-operator method (plans C and D) allows to reduce the execution time drastically. Especially, it allows to reduce

the complexity of the join operation from quadratic to near linear complexity, O(N *log(N)), as can be seen for plan D. In plan D nearly all execution time is needed for the processing of the exact geometries.

Table 2: Performance results of different sequential plans of the test query (time in sec.s)

sequential execution of plan	table PG1	table PG 2	table PG3
A	13582.6	not avail.	not avail.
B	430.9	41663.1	not avail.
C	50.8	3388.8	not avail.
D[a]	17.6	189.6	1908.6
D (time without refinement)	0.6	6.5	96.2
D (time for refinement only)	17.0	183.1	1812.4

a. Please note that for plan D the results shown in Table 2 are for the spatial partitioning that yielded the lowest execution time from Table 3.

Given the times for exact geometry processing, we can see that the overall execution times for plans B and C still grow quadratically (i.e. they grow by a factor 100 when we move to a table that has 10 times more tuples), if the overall execution time is dominated by the join costs. Since plan D uses spatial partitioning, the asymptotic complexity of the plan is determined by the merge join, i.e. it is only O(N *log(N)). Thus the increase in the execution times is much slower. Since the total execution time of plan D for the test tables is dominated by the time for refinement, the overall increase of the execution time is roughly linear.

Table 3 demonstrates directly that the spatial partitioning allows to reduce the complexity of the join operation from quadratic to near linear complexity. It shows the effect of different numbers of buckets for the spatial partitioning on the number of replicated polygons and the execution time for plan D. In addition we computed the execution time minus the time for refinement. As can be seen, the execution time decreases, until the number of buckets approaches the number of polygons in the table. The execution time of plan D without refinement is reduced roughly linearly with an increasing number of buckets, until the number of polygons per bucket is small. This is what we expected, as corresponding buckets are joined with a nested-loops join. Therefore, if the number of buckets is increased by a factor K, the time to join two buckets is reduced by K^2, since the number of tuples in a bucket is reduced by a factor of about K. Only if more buckets than polygons are used and the number of replicated polygons increases, the overall execution time increases slightly. This is not surprising, since most of the time is needed for exact geometry processing in these cases.

In Table 4 we present the effect of parallelism on the execution time. We executed plans B, C, and D with a degree of parallelism of 2 for the nested-loops/merge join and the restriction by `exact_overlaps`. Parallelization could be achieved as usual, i.e. for the nested-loops join by splitting one input into 2 partitions and by replicating the other input and for the merge join by splitting both inputs into 2 partitions using the join attribute (i.e. the bucket number). Then the restriction with

Table 3: Impact of number of buckets and replication on performance of plan D (response time in sec.s)

# buckets	PG1: number of replicated polygons	PG1: execution time for plan D	PG1: exec. time for plan D without refinement	PG 2: number of replicated polygons	PG 2: execution time for plan D	PG 2: exec. time for plan D without refinement	PG 3: number of replicated polygons	PG3: execution time for plan D	PG3: exec. time for plan D without refinement
1	0	53.9	36.9	0	3 696.8	3 513.7	0	not avail.	not avail.
4	11	26.5	9.5	38	1 073.5	890.4	143	90 172.8	88 360.4
16	37	19.6	2.6	125	412.9	229.8	423	24 078.4	22 266.0
64	100	17.9	0.9	310	245.2	60.6	973	7 402.7	5 590.3
256	221	17.6	0.6	678	202.7	19.6	2 107	3 317.2	1 504.8
1 024	489	17.6	0.6	1 445	192.1	9.0	4 419	2 311.1	498.7
4 096	1 132	17.6	0.6	3 080	189.6	6.5	9 100	2 037.1	224.7
16 384	2 628	17.9	0.9	6 652	189.7	6.6	18 904	1 962.0	149.6
65 536	7 155	18.7	2.7	15 308	191.8	8.7	39 975	1 908.6	96.2
262 144	21 666	21.0	4.0	38 249	192.5	9.4	87 068	1 996.0	183.6

`exact_overlaps` was evaluated on both resulting partitions. Since the execution is always CPU bound and we used all processors of our machine, the maximum obtainable speedup is slightly less than two, because the operating system uses some CPU resources. For sequential execution this operating system overhead can be handled by the CPU that is not used for query processing. As one can see from the speedup (shown in the shaded rows), all plans were able to profit from parallel execution, as we expected, since the execution is CPU bound. The speedup is decreasing from plan B to D, since the execution becomes less CPU-bound and I/O parallelism was not fully exploited during these tests. We did not evaluate plan A in parallel, as this plan is anyway not usable.

These initial performance results show clearly the impressive performance gains that are possible with the multi-operator method (plan D compared to the single-operator implementation plan B). In addition, the results show that parallelism can be applied to the multi-operator implementations in the usual way to speed up processing further, i.e. by means of data parallelism and pipelining.

6. Related work

User-Defined Functions (UDFs) have attracted increasing interest of researchers as well as the industry in recent years (see e.g. [1], [7], [15], [16], [24], [27], [32], [34], [38], [40]). However, most of the work discusses only the non-parallel execution of UDFs, special implementation techniques like caching, or it is directed towards query optimization for UDFs. In [34] pipeline parallelism for functions as well as intra-function parallelism are proposed, but no details of a framework for the implementation of these concepts are given. In [35] support for the parallel implementation of ADTs and UDFs in the area of geo-spatial applications is discussed. It is remarked that in this area complex multi-step operations

Table 4: Performance results for parallel processing of plans B, C, and D (time in sec.s)

plan	degree of parallelism	table PG1	table PG 2	table PG3
B	1	430.9	41 663.1	not avail.
B	2	218.6	22 129.4	not avail.
speedup	-	1.97	1.88	not avail.
C	1	50.8	3 388.8	not avail.
C	2	28.0	1 848.4	not avail.
speedup	-	1.81	1.83	not avail.
D	1	17.6	189.6	2 037.1
D	2	11.2	111.1	1 094.3
speedup	-	1.57	1.70	1.86

are commonplace. Also special new join techniques [36] and other special implementation techniques have been proposed, but no approach that allows the execution of such special techniques in current ORDBMS was mentioned.

In [24] E-ADTs are proposed as a new approach for the architecture of ORDBMS. An ORDBMS is envisioned as a collection of E-ADTs (enhanced ADTs). These E-ADTs encapsulate the complete functionality and implementation of the ADTs. Especially all knowledge about the query language, its optimization and the execution engine are encapsulated in the E-ADT. All E-ADT implementations are linked via common interfaces. Furthermore the implementation of new E-ADTs can use a common library with base services, e.g. for storage management. We believe that this is an interesting approach that is in general more ambitious than the multi-operator method discussed here, but parallel execution is not examined for E-ADTs. Basically it should be possible to use multi-operator implementations also within the E-ADT approach. While the multi-operator method might be useful for the E-ADT approach, in addition and in contrast to the E-ADT approach it fits very well to the architectures of current commercial ORDBMS.

In [23] a framework for parallel processing of user-defined scalar and aggregate functions in ORDBMS is proposed. This framework allows to specify for a given operator, whether data parallelism can be used for evaluation. If data parallelism is applicable, the developer has to specify, which partitioning functions are allowed. For aggregate functions, an implementation in two plan operators is proposed to support parallelism (cf. [38] for a similar approach). Additionally, the possibility to specify sorting as a preprocessing step is introduced. While these measures provide some limited support for the parallel execution of user-defined functions that have a global context (like aggregation), the multi-operator method supports efficient and parallel execution for much more user-defined functions. Therefore the multi-operator method should be seen as a further generalization and important supplementation of [23]: The possibility to use several operators allows the developer to separate different parts of the implementation. This will often allow to use data parallelism for the individual parts, though a data par-

allel execution might not be possible for a complex single-operator implementation with external effects or a global state.

7. Summary

In this paper we have proposed the multi-operator method as a new technique for the integration of algorithms for the efficient evaluation of user-defined functions. The goal of this method is to provide a general implementation method that enables developers to implement efficient algorithms for complex UDPs and execute them in ORDBMS. The main advantage of the multi-operator method is that in order to support new database operations it is only necessary to write a few UDFs and place them in a database operator graph. The alternative for developers would be to write a new specialized database operator (a much more difficult task). However, this is not possible in current ORDBMS and currently there seems also to be no appropriate technology for this. Hence the multi-operator method is in our view a feasible and practical method that can be used without changes of the current ORDBMS core systems. The multi-operator method offers some important further advantages: the finer granularity of the implementation structure enhances *reuse*, *optimization,* and especially *parallelization*. This support for parallel execution is a significant advantage of the multi-operator method.

An important application of the multi-operator method are user-defined join algorithms that allow highly efficient implementations of UDPs as our performance measurements have demonstrated. We are not aware of any other approach to the user-defined implementation of joins that fits well to the current technology of parallel ORDBMS.

Acknowledgments

We gratefully acknowledge the help of our colleagues Clara Nippl and Stephan Zimmermann with the prototype. We would also like to acknowledge the implementation support by S. Heupel and R. Schumacher. The feedback from C. Mohan, G. Lohman and M. Carey on the issue of interfaces for query execution plans is gratefully acknowledged.

References

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

[2] Apers, P. M. G., van den Berg, C. A., Flokstra, J., Grefen, P. W. P. J., Kersten, M. L., Wilschut, A. N.: PRISMA /DB: A Parallel Main Memory Relational DBMS. TKDE 4(6): 541-554 (1992).

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

[4] Brinkhoff, T., Kriegel, H.-P., Schneider, R., Seeger, B.: Multi-Step Processing of Spatial Joins. SIGMOD Conf. 1994: 197-208.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[20] Informix Universal Server, Virtual-Index Interface Programmer's Manual, Vers. 9.1, Informix Software Inc., 1997.

[21] Informix Universal Server, Guide to the Virtual Table Interface, Vers. 9.0, Informix Software Inc., 1996.

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

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

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

[25] Lohman, G. M.: Grammar-like Functional Rules for Representing Query Optimization Alternatives. SIGMOD Conf. 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] Mishra, P., Eich, M. H.: Join Processing in Relational Databases. Computing Surveys 24(1): 63-113 (1992).

[29] Mitschang, B.: Query Processing in Database Systems (in german), Vieweg, 1995.

[30] Näher, S., Uhrig, C.: The LEDA User Manual, Version R 3.5, 1997 (http://www.mpi-sb.mpg.de/LEDA/leda.html).

[31] Nippl, C., Mitschang, B.: TOPAZ: a Cost-Based, Rule-Driven, Multi-Phase Parallelizer, VLDB 1998: 251-262.

[32] Nori, A., Kumar, S.: Bringing Objects to the Mainstream, COMPCON Conference, 1997: 136-142.

[33] 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.

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

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

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

[37] 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 Conf. 1997: 336-347.

[38] Roy, J.: Aggregations in the Environment of Informix Dynamic Server with Universal data Option. Informix Tech Notes, Volume 8, Issue 2, Informix Software Inc., 1998.

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

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

[41] TransBase Relational Database System, Version 3.3, System Guide, TransAction SoftwareGmbH, Munich, 1988.

[42] Zhou, X., Abel, D. J., Truffet, D.: Data Partitioning for Parallel Spatial Join Processing. SSD 1997: 178-196.

SFB 342: Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

bisher erschienen :

Reihe A

Liste aller erschienenen Berichte von 1990-1994
auf besondere Anforderung

342/01/95 A Hans-Joachim Bungartz: Higher Order Finite Elements on Sparse Grids

342/02/95 A Tao Zhang, Seonglim Kang, Lester R. Lipsky: The Performance of Parallel Computers: Order Statistics and Amdahl's Law

342/03/95 A Lester R. Lipsky, Appie van de Liefvoort: Transformation of the Kronecker Product of Identical Servers to a Reduced Product Space

342/04/95 A Pierre Fiorini, Lester R. Lipsky, Wen-Jung Hsin, Appie van de Liefvoort: Auto-Correlation of Lag-k For Customers Departing From Semi-Markov Processes

342/05/95 A Sascha Hilgenfeldt, Robert Balder, Christoph Zenger: Sparse Grids: Applications to Multi-dimensional Schrödinger Problems

342/06/95 A Maximilian Fuchs: Formal Design of a Model-N Counter

342/07/95 A Hans-Joachim Bungartz, Stefan Schulte: Coupled Problems in Microsystem Technology

342/08/95 A Alexander Pfaffinger: Parallel Communication on Workstation Networks with Complex Topologies

342/09/95 A Ketil Stølen: Assumption/Commitment Rules for Data-flow Networks - with an Emphasis on Completeness

342/10/95 A Ketil Stølen, Max Fuchs: A Formal Method for Hardware/Software Co-Design

342/11/95 A Thomas Schnekenburger: The ALDY Load Distribution System

342/12/95 A Javier Esparza, Stefan Römer, Walter Vogler: An Improvement of McMillan's Unfolding Algorithm

342/13/95 A Stephan Melzer, Javier Esparza: Checking System Properties via Integer Programming

342/14/95 A Radu Grosu, Ketil Stølen: A Denotational Model for Mobile Point-to-Point Dataflow Networks

342/15/95 A Andrei Kovalyov, Javier Esparza: A Polynomial Algorithm to Compute the Concurrency Relation of Free-Choice Signal Transition Graphs

342/16/95 A Bernhard Schätz, Katharina Spies: Formale Syntax zur logischen Kernsprache der Focus-Entwicklungsmethodik

342/17/95 A Georg Stellner: Using CoCheck on a Network of Workstations

342/18/95 A Arndt Bode, Thomas Ludwig, Vaidy Sunderam, Roland Wismüller: Workshop on PVM, MPI, Tools and Applications

342/19/95 A Thomas Schnekenburger: Integration of Load Distribution into ParMod-C

342/20/95 A Ketil Stølen: Refinement Principles Supporting the Transition from Asynchronous to Synchronous Communication

Reihe A

342/21/95 A Andreas Listl, Giannis Bozas: Performance Gains Using Subpages for Cache Coherency Control

342/22/95 A Volker Heun, Ernst W. Mayr: Embedding Graphs with Bounded Treewidth into Optimal Hypercubes

342/23/95 A Petr Jančar, Javier Esparza: Deciding Finiteness of Petri Nets up to Bisimulation

342/24/95 A M. Jung, U. Rüde: Implicit Extrapolation Methods for Variable Coefficient Problems

342/01/96 A Michael Griebel, Tilman Neunhoeffer, Hans Regler: Algebraic Multigrid Methods for the Solution of the Navier-Stokes Equations in Complicated Geometries

342/02/96 A Thomas Grauschopf, Michael Griebel, Hans Regler: Additive Multilevel-Preconditioners based on Bilinear Interpolation, Matrix Dependent Geometric Coarsening and Algebraic-Multigrid Coarsening for Second Order Elliptic PDEs

342/03/96 A Volker Heun, Ernst W. Mayr: Optimal Dynamic Edge-Disjoint Embeddings of Complete Binary Trees into Hypercubes

342/04/96 A Thomas Huckle: Efficient Computation of Sparse Approximate Inverses

342/05/96 A Thomas Ludwig, Roland Wismüller, Vaidy Sunderam, Arndt Bode: OMIS — On-line Monitoring Interface Specification

342/06/96 A Ekkart Kindler: A Compositional Partial Order Semantics for Petri Net Components

342/07/96 A Richard Mayr: Some Results on Basic Parallel Processes

342/08/96 A Ralph Radermacher, Frank Weimer: INSEL Syntax-Bericht

342/09/96 A P.P. Spies, C. Eckert, M. Lange, D. Marek, R. Radermacher, F. Weimer, H.-M. Windisch: Sprachkonzepte zur Konstruktion verteilter Systeme

342/10/96 A Stefan Lamberts, Thomas Ludwig, Christian Röder, Arndt Bode: PFS-Lib – A File System for Parallel Programming Environments

342/11/96 A Manfred Broy, Gheorghe Ştefănescu: The Algebra of Stream Processing Functions

342/12/96 A Javier Esparza: Reachability in Live and Safe Free-Choice Petri Nets is NP-complete

342/13/96 A Radu Grosu, Ketil Stølen: A Denotational Model for Mobile Many-to-Many Data-flow Networks

342/14/96 A Giannis Bozas, Michael Jaedicke, Andreas Listl, Bernhard Mitschang, Angelika Reiser, Stephan Zimmermann: On Transforming a Sequential SQL-DBMS into a Parallel One: First Results and Experiences of the MIDAS Project

342/15/96 A Richard Mayr: A Tableau System for Model Checking Petri Nets with a Fragment of the Linear Time μ -Calculus

342/16/96 A Ursula Hinkel, Katharina Spies: Anleitung zur Spezifikation von mobilen, dynamischen Focus-Netzen

342/17/96 A Richard Mayr: Model Checking PA-Processes

342/18/96 A Michaela Huhn, Peter Niebert, Frank Wallner: Put your Model Checker on Diet: Verification on Local States

342/01/97 A Tobias Müller, Stefan Lamberts, Ursula Maier, Georg Stellner: Evaluierung der Leistungsfähigkeit eines ATM-Netzes mit parallelen Programmierbibliotheken

Reihe A

342/02/97 A Hans-Joachim Bungartz and Thomas Dornseifer: Sparse Grids: Recent Developments for Elliptic Partial Differential Equations

342/03/97 A Bernhard Mitschang: Technologie für Parallele Datenbanken - Bericht zum Workshop

342/04/97 A nicht erschienen

342/05/97 A Hans-Joachim Bungartz, Ralf Ebner, Stefan Schulte: Hierarchische Basen zur effizienten Kopplung substrukturierter Probleme der Strukturmechanik

342/06/97 A Hans-Joachim Bungartz, Anton Frank, Florian Meier, Tilman Neunhoeffer, Stefan Schulte: Fluid Structure Interaction: 3D Numerical Simulation and Visualization of a Micropump

342/07/97 A Javier Esparza, Stephan Melzer: Model Checking LTL using Constraint Programming

342/08/97 A Niels Reimer: Untersuchung von Strategien für verteiltes Last- und Ressourcenmanagement

342/09/97 A Markus Pizka: Design and Implementation of the GNU INSEL-Compiler gic

342/10/97 A Manfred Broy, Franz Regensburger, Bernhard Schätz, Katharina Spies: The Steamboiler Specification - A Case Study in Focus

342/11/97 A Christine Röckl: How to Make Substitution Preserve Strong Bisimilarity

342/12/97 A Christian B. Czech: Architektur und Konzept des Dycos-Kerns

342/13/97 A Jan Philipps, Alexander Schmidt: Traffic Flow by Data Flow

342/14/97 A Norbert Fröhlich, Rolf Schlagenhaft, Josef Fleischmann: Partitioning VLSI-Circuits for Parallel Simulation on Transistor Level

342/15/97 A Frank Weimer: DaViT: Ein System zur interaktiven Ausführung und zur Visualisierung von INSEL-Programmen

342/16/97 A Niels Reimer, Jürgen Rudolph, Katharina Spies: Von FOCUS nach INSEL - Eine Aufzugssteuerung

342/17/97 A Radu Grosu, Ketil Stølen, Manfred Broy: A Denotational Model for Mobile Point-to-Point Data-flow Networks with Channel Sharing

342/18/97 A Christian Röder, Georg Stellner: Design of Load Management for Parallel Applications in Networks of Heterogenous Workstations

342/19/97 A Frank Wallner: Model Checking LTL Using Net Unfoldings

342/20/97 A Andreas Wolf, Andreas Kmoch: Einsatz eines automatischen Theorembeweisers in einer taktikgesteuerten Beweisumgebung zur Lösung eines Beispiels aus der Hardware-Verifikation – Fallstudie –

342/21/97 A Andreas Wolf, Marc Fuchs: Cooperative Parallel Automated Theorem Proving

342/22/97 A T. Ludwig, R. Wismüller, V. Sunderam, A. Bode: OMIS - On-line Monitoring Interface Specification (Version 2.0)

342/23/97 A Stephan Merkel: Verification of Fault Tolerant Algorithms Using PEP

342/24/97 A Manfred Broy, Max Breitling, Bernhard Schätz, Katharina Spies: Summary of Case Studies in Focus - Part II

342/25/97 A Michael Jaedicke, Bernhard Mitschang: A Framework for Parallel Processing of Aggregat and Scalar Functions in Object-Relational DBMS

342/26/97 A Marc Fuchs: Similarity-Based Lemma Generation with Lemma-Delaying Tableau Enumeration

Reihe A

342/27/97 A Max Breitling: Formalizing and Verifying TimeWarp with FOCUS

342/28/97 A Peter Jakobi, Andreas Wolf: DBFW: A Simple DataBase FrameWork for the Evaluation and Maintenance of Automated Theorem Prover Data (incl. Documentation)

342/29/97 A Radu Grosu, Ketil Stølen: Compositional Specification of Mobile Systems

342/01/98 A A. Bode, A. Ganz, C. Gold, S. Petri, N. Reimer, B. Schiemann, T. Schnekenburger (Herausgeber): "Anwendungsbezogene Lastverteilung", ALV'98

342/02/98 A Ursula Hinkel: Home Shopping - Die Spezifikation einer Kommunikationsanwendung in FOCUS

342/03/98 A Katharina Spies: Eine Methode zur formalen Modellierung von Betriebssystemkonzepten

342/04/98 A Stefan Bischof, Ernst W. Mayr: On-Line Scheduling of Parallel Jobs with Runtime Restrictions

342/05/98 A St. Bischof, R. Ebner, Th. Erlebach: Load Balancing for Problems with Good Bisectors and Applications in Finite Element Simulations: Worst-case Analysis and Practical Results

342/06/98 A Giannis Bozas, Susanne Kober: Logging and Crash Recovery in Shared-Disk Database Systems

342/07/98 A Markus Pizka: Distributed Virtual Address Space Management in the MoDiS-OS

342/08/98 A Niels Reimer: Strategien für ein verteiltes Last- und Ressourcenmanagement

342/09/98 A Javier Esparza, Editor: Proceedings of INFINITY'98

342/10/98 A Richard Mayr: Lossy Counter Machines

342/11/98 A Thomas Huckle: Matrix Multilevel Methods and Preconditioning

342/12/98 A Thomas Huckle: Approximate Sparsity Patterns for the Inverse of a Matrix and Preconditioning

342/13/98 A Antonin Kucera, Richard Mayr: Weak Bisimilarity with Infinite-State Systems can be Decided in Polynomial Time

342/01/99 A Antonin Kucera, Richard Mayr: Simulation Preorder on Simple Process Algebras

342/02/99 A Johann Schumann, Max Breitling: Formalisierung und Beweis einer Verfeinerung aus FOCUS mit automatischen Theorembeweisern – Fallstudie

342/03/99 A M. Bader, M. Schimper, Chr. Zenger: Hierarchical Bases for the Indefinite Helmholtz Equation

342/04/99 A Frank Strobl, Alexander Wisspeintner: Specification of an Elevator Control System

SFB 342 : Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

Reihe B

342/1/90 B Wolfgang Reisig: Petri Nets and Algebraic Specifications

342/2/90 B Jörg Desel: On Abstraction of Nets

342/3/90 B Jörg Desel: Reduction and Design of Well-behaved Free-choice Systems

342/4/90 B Franz Abstreiter, Michael Friedrich, Hans-Jürgen Plewan: Das Werkzeug runtime zur Beobachtung verteilter und paralleler Programme

342/1/91 B Barbara Paech: Concurrency as a Modality

342/2/91 B Birgit Kandler, Markus Pawlowski: SAM: Eine Sortier-Toolbox – Anwenderbeschreibung

342/3/91 B Erwin Loibl, Hans Obermaier, Markus Pawlowski: 2. Workshop über Parallelisierung von Datenbanksystemen

342/4/91 B Werner Pohlmann: A Limitation of Distributed Simulation Methods

342/5/91 B Dominik Gomm, Ekkart Kindler: A Weakly Coherent Virtually Shared Memory Scheme: Formal Specification and Analysis

342/6/91 B Dominik Gomm, Ekkart Kindler: Causality Based Specification and Correctness Proof of a Virtually Shared Memory Scheme

342/7/91 B W. Reisig: Concurrent Temporal Logic

342/1/92 B Malte Grosse, Christian B. Suttner: A Parallel Algorithm for Set-of-Support
Christian B. Suttner: Parallel Computation of Multiple Sets-of-Support

342/2/92 B Arndt Bode, Hartmut Wedekind: Parallelrechner: Theorie, Hardware, Software, Anwendungen

342/1/93 B Max Fuchs: Funktionale Spezifikation einer Geschwindigkeitsregelung

342/2/93 B Ekkart Kindler: Sicherheits- und Lebendigkeitseigenschaften: Ein Literaturüberblick

342/1/94 B Andreas Listl, Thomas Schnekenburger, Michael Friedrich: Zum Entwurf eines Prototypen für MIDAS