Parallel Query Support for Multidimensional Data: Inter-object Parallelism◆

Karl Hahn[1], Bernd Reiner[1], Gabriele Höfling[1], Peter Baumann[2]

[1] FORWISS, Bavarian Research Center for Knowledge Based Systems, Munich, Germany
{hahnk,reiner,hoefling}@forwiss.de
http://www.wibas.forwiss.tu-muenchen.de

[2] Active Knowledge GmbH, Munich, Germany
baumann@active-knowledge.com
http://www.active-knowledge.de

Abstract. Intra-query parallelism is a well-established mechanism for achieving high performance in (object-) relational database systems. However, the methods have yet not been applied to the upcoming field of multidimensional array databases. Specific properties of multidimensional array data require new parallel algorithms. This paper presents a number of new techniques for parallelizing queries in multidimensional array database management systems. It discusses their implementation in the RasDaMan DBMS, the first DBMS for generic multidimensional array data. The efficiency of the techniques presented is demonstrated using typical queries on large multidimensional data volumes.

1 Introduction

Arrays of arbitrary size and dimensionality appear in a large variety of database application fields, e.g., medical imaging, geographic information systems [7], scientific simulations, etc. Recently, integration of an application domain-independent and of a generic type constructor for such *Multidimensional Discrete Data (MDD)* into *Database Management Systems (DBMS)* has received growing attention. Current scientific contributions in this area mainly focus on MDD algebra and specialized storage architectures [1] [2] [3].

Since MDD objects may have a magnitude of several MB and much more and, compared to scalar values, operations on these values can be very complex, their efficient evaluation becomes a critical factor for the overall query response time. Beyond query optimization, parallel query processing is the most promising technique to speed up complex operations on large data volumes.

One of the outcomes of the predecessor project of *ESTEDI* (http://www.estedi.org), called *RasDaMan* (funded by the European Commission under grant no. 20073), in

◆ This work was supported by the ESTEDI project (http://www.estedi.org). ESTEDI (European Spatio-Temporal Data Infrastructure for High-Performance Computing) is funded by the European Commission under FP5 grant no. IST-1999-11009.

which the *Array DBMS RasDaMan* [2] has been developed, was the awareness that most queries on multidimensional array data are in fact CPU-bound [10]. Therefore, one major research issue of the succeeding project *ESTEDI* is the parallel processing of queries which is the topic of this paper. Furthermore, ESTEDI, an initiative of European software vendors and supercomputing centers, will establish an European standard for the storage and retrieval of multidimensional high-performance computing (HPC) data. It addresses a main technical obstacle, the delivery bottleneck of large HPC results to the users, by augmenting high-volume data generators with a flexible data management and extraction tool for multidimensional array data.

This paper discusses the suitability of concepts developed in parallel relational DBMS for intra-query parallelism in array DBMS. Special properties of array data, e.g. the size of one single data object combined with expensive cell operations require adapted algorithms for parallel processing. Suitable concepts found in relational DBMS were implemented and evaluated in the RasDaMan Array DBMS.

The remainder of this paper is organized as follows. Section 2 briefly describes the multidimensional data model, the multidimensional query language RasQL and the query execution in our example Array DBMS RasDaMan. In section 3, the architecture of the parallel RasDaMan server and the parallelizer module, which rewrites the query tree in order to distribute different sections of the tree to different processes, will be presented. The performance of the parallel implementation will be discussed, using a running example in section 4. We finally compare parallel algorithms of relational systems to our implemented techniques in order to evaluate their suitability regarding array data. Section 5 contains our conclusions and suggestions for future work.

2 Processing Multidimensional Data: the Array DBMS RasDaMan

In this section, we will describe a multidimensional data model, a multidimensional query language and the execution of multidimensional queries. As the parallel query processing was implemented in RasDaMan [2], we will first introduce the RasDaMan data model, the RasQL query language and its internal query tree and query execution. Nevertheless, the concepts for parallel query processing on array DBMS described in section 3 can be applied to other array DBMS as well.

2.1 Logical Data Model and Query Language

The fundamental concept of the RasDaMan data model is *Multidimensional Discrete Data (MDD).* This can be defined as multidimensional array with (1) an arbitrary dimensionality, (2) a spatial domain, specified via lower bounds and upper bounds for each dimension, (3) a specific cell base type, consisting of a single scalar value or a complex type structure. An *MDD collection* holds an unordered set of MDD with the same dimensionality, spatial domain and cell base type.

In Fig. 2 (left top) we see a 3D MDD (data based on climate simulation model, provided by Max-Planck Institute for Meteorology, one of the application partners in the

the ESTEDI project). The dimensions specify longitude, latitude and time (months). The spatial domain is [0:63, 0:127, 0:119], i.e. the 3-dimensional array includes 64 x 128 x 120 cells. The cell values of scalar type double have a range of about 200 (dark regions near the poles) to 320 (bright regions near the equator), and define average temperatures on the earth surface in degrees Kelvin for 120 months. In Fig. 2 (left bottom), a collection of five MDD (each representing a decade of average temperature values) with the same dimensionality, spatial domain and cell type is illustrated. Internally each MDD is identified by a unique object identification number, here 28 to 32.

In order to invoke operations on array data and specify the multidimensional interval to be accessed, RasDaMan provides a query language RasQL which is derived from standard SQL. The simplified structure of such a RasQL query is

```
SELECT <array operation>
FROM collection 1, ..., collection n
WHERE <condition operation>
```

Array operations used in the select and where clause of the statement can be

1. geometric operations: the trimming operation specifies a sub-array with the same dimensionality, e.g. in our example [0:63,0:127,0:11] for the first 12 months. A section operation reduces the dimensionality by one, i.e. the data is projected to a hyperplane.
2. induced operations: operations which are defined on the base cell type are also defined on multidimensional arrays, e.g. the sum of two MDD with cell base type double.
3. aggregation operations: an MDD is reduced to one single scalar value. Operations of this class are quantifiers, maximum, minimum, average, etc.

The primary benefit of such a complex query language is the minimization of data transmission between database server and client. Areas of interest can be specified with geometric operators, and complex calculations can be executed on the server side, only transferring the result to the client instead of entire objects. A more detailed specification of the RasDaMan data model and the RasQL query language can be found in [2] and [3].

2.2 Query Execution: the RasDaMan Query Tree

Internally, the RasDaMan server builds up a query tree to process the query (Fig. 1). The query execution follows the open-next-close protocol (iterator concept) which is well known in database technology. First, the method `open()` is invoked on the root node α. In a post-order traversal, the method invocation is propagated through the query tree while initializing resources. Then, method `next()` is invoked repeatedly on the root node which again is propagated in a post-order traversal through the entire tree. Each time the method completes, this bottom-up process returns one element of the result collection. At the end, method `close()` is called to clean up resources allocated during execution.

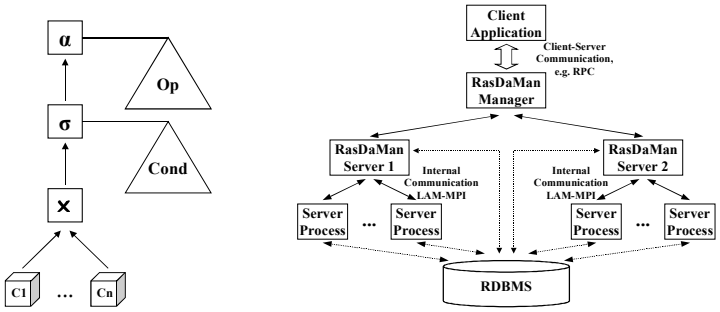

Fig. 1. Structure of a query tree for a RasQL statement (left).
Architecture of the parallel RasDaMan server (right).

The iterator nodes of the query tree using the open-next-close protocol in Fig. 1 are

1. cross product `x`, representing the FROM clause of the RasQL statement. It delivers the cross product of all multidimensional objects of all referenced collections.
2. selection σ, representing the WHERE condition of the RasQL statement. The condition tree consists of multidimensional operations. A `next()` returns the next multidimensional data object for which the condition tree evaluates true.
3. application α, representing the SELECT operation of the RasQL statement. The operation tree executes array operations on the resulting multidimensional data.

It should be pointed out that the multidimensional data is not loaded at the bottom of the query tree (like the relational scan operator) but demand-driven during the evaluation of the condition tree, evaluated by the selection σ, and the operation tree, evaluated by the application α. Therefore, these operations of the query tree are the most expensive ones, because on the one hand the data loading is done there, and on the other hand the operation and condition itself are trees, which represent expensive array operations on the multidimensional data. Details on query optimization and execution in array DBMS can be found in [10].

3 Parallel Query Processing

Parallel query processing is a well established mechanism in relational DBMS [4] [6] [9] [11]. Different hardware architectures have been investigated regarding parallelism, i.e. multiprocessor computer (shared everything, symmetric multiprocessing, SMP) and shared disc / shared nothing systems (e.g. workstation cluster). Considering the facts that queries on array data are in most cases highly CPU-bound [10] and intermediate results can reach a size of several MB and more, we came to the conclusion that shared everything architectures are more appropriate for a parallel array DBMS, as performance will decrease with the transmission of large intermediate results over a network. Nevertheless, the architecture and the communication protocol used by the parallel RasDaMan server, which is LAM-MPI (http://www.lam-mpi.org), is not limited to any specific hardware architecture. LAM (Local Area Multicomputer) is an

MPI (Message Passing Interface, http://www.mcs.anl.gov/mpi) programming environment and development system for heterogeneous computers within a network. With LAM, a dedicated cluster or an existing network computing infrastructure can act as one parallel computer solving one problem.

Furthermore, parallel processing of data can be classified into data parallelism, where different data sets are handled by different processes, and pipeline parallelism, where we can utilize a producer-consumer relationship within the query tree (a consumer executes operations on a data stream which is still being generated by the producer process). In this section we will describe the implemented data parallelism; a further discussion of the adaptation of parallel techniques in relational DBMS (e.g. pipeline parallelism) to Array DBMS will follow in section 4.

3.1 Parallel Architecture

The overall process structure of the parallel RasDaMan array DBMS is shown in Fig. 1. Several types of client applications currently exist. The most important is rView, a visualization application provided with RasDaMan [5]. Furthermore, several graphical front-ends for the visualization of data were implemented in the ESTEDI project, e.g., in the field of gene expression simulation, meteorological simulations, satellite image retrieval and information extraction, flow modeling of chemical reactors, etc. The clients are connected to the RasDaMan Manager using the RPC[1] or HTTP[2] protocol.

The RasDaMan manager distributes client request to different RasDaMan server processes which typically run on different computers. On this level, inter-query parallelism (multi-user functionality) is achieved: a query sent by the client application is transferred from the manager to a server process that is currently available.

Each RasDaMan server itself forks several internal processes at start-up time to realize intra-query parallelism. One designated process is responsible for the client-server communication and the distribution of the workload, all other processes are internal and therefore not visible from the client. The internal server processes have access to a relational DBMS which acts as a storage and transaction manager for the multidimensional array data (dotted lines in Fig. 1). At the time of writing, supported relational DBMS are Oracle, IBM DB2 and Informix.

As mentioned above, distributed processing of RasQL queries requires different processes and communication between them, e.g. to exchange requests or intermediate results. In order to avoid performance problems while evaluating a query, the processes do not fork during query execution but at start-up time of a RasDaMan server. So whenever a RasDaMan server is started, we create several internal server processes which reside in memory and are waiting for requests. We run 2 processes for administration tasks and an arbitrary number of processes for the computational work. In order to utilize CPU resources, but avoid unnecessary swapping of processes, we recommend n+2 processes with n being the number of processors of an SMP computer, resp. the number of nodes in a workstation cluster.

---

[1] RPC: Remote Procedure Call

[2] HTTP: Hypertext Transfer Protocol

The parallel RasDaMan server distinguishes 3 classes of processes, where each process class reflects a part of the overall query tree (Fig. 1) that can be executed independently from other processes:

1. RasDaMan master server: this process is responsible for the server client communication via network, e.g., it connects the RasDaMan server to the rView client application using the RPC or HTTP protocol. It distributes controlling messages, the queries and therefore the workload to all other (internal) server processes. Query results are collected from the internal processes and transmitted to the client.
2. Internal tuple server: this process generates the cross-product of all MDD involved in the query. This is required to ensure a central administration of the multidimensional data objects for all processes. Upon receiving a request for the next data element, the server process accesses the underlying relational DBMS (only object identifiers are read not the whole objects), and sends the next tuple of object identifiers to the calling process.
3. Internal worker processes: a number of processes which do the actual query processing. Receiving a data identifier from the tuple server (invocation of `next()`) these processes evaluate the condition tree and the operation tree on this (tuple of) MDD.

3.2 Parallel Query Tree

The classification of the internal server processes presented above corresponds with the structure of the (parallel) query tree. In relational DBMS the optimization and parallelization phases, which both restructure the query tree, often show interferences. The optimization of a sequential execution plan contradicts an optimal parallel execution plan. Therefore, optimization and parallelization phases are often combined in one single module [8]. In RasDaMan, the optimization of the query tree does not conflict with parallel optimization, as optimization is primarily performed on the array operations of the operation and condition tree while the parallelizer works on the iterator nodes, so the parallelization module is invoked after the optimization phase.

The RasDaMan parallelization module works as follows: the algorithm identifies cross product iterator nodes in the query tree and inserts a pair of send/receive[3] nodes above it. The send/receive nodes encapsulate the transmission of requests (query execution protocol) and intermediate results between the internal server processes, using the MPI protocol. On the top of the query tree, the algorithm inserts another pair of send/receive nodes. This is necessary to designate one process as the master server process which also handles the server-client communication. It should be noted that both ‘expensive’ nodes, the application and the selection nodes which evaluate the array operations, are executed within one process, instead of being split into two processes. We decided not to compute the application node and the selection node on different processes because this would avoid the usage of transient multidimensional data, i.e. memory-cached objects of the data read from the relational DBMS. It is very

[3] The semantics of the send and receive nodes refers to the direction of messages containing (intermediate) results.

typical to execute selection and application operation on the same sub-arrays of data, therefore RasDaMan caches intermediate results within a query.

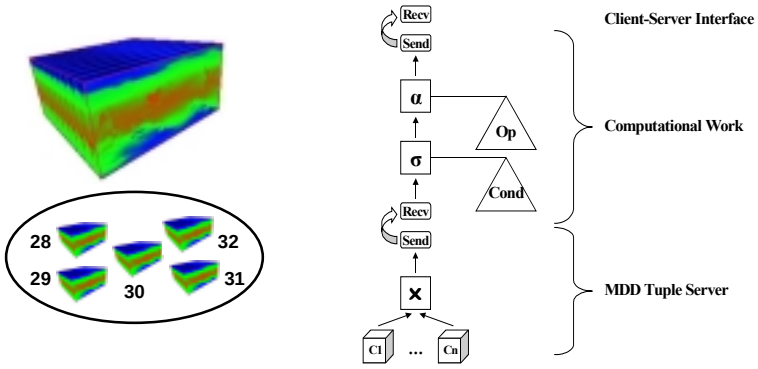

Fig. 2. 3-dimensional MDD (left top). Collection of five MDD (left bottom). Parallel query tree (right).

The algorithm splits the query tree into several sections which are interconnected by pairs of send/receive nodes. Fig. 2 illustrates an example of a query tree which was prepared for parallel query processing by the parallelizer module. The query tree is split into three parts which are processed by the different process classes presented above. The master process only holds one single receive node on the top of the query tree. It distributes the query execution, i.e. the open-next-close iterator concept, top-down in the query tree to the internal worker processes and collects the query results returned. The complete result is then encapsulated in a transfer structure and transmitted to the client.

Application and selection nodes of the query tree represent the expensive operations performed on the array data. This computational workload is distributed between several worker processes. Each internal worker process executes the query tree with the upper send node as its root node. Whenever it receives a `next()` request from the master process via MPI, it sends a next request down the query tree to the internal tuple server process and receives the MDD identifiers on which the operations are then evaluated. The resulting data is transmitted to the master which collects all results and transmits them to the client.

As mentioned above, the tuple server process executes the query tree beginning at the lower send node. It delivers the next valid identifier for a tuple of MDD, accessing the referenced collections. With this concept, a central allocation of MDD tuples to the worker processes is assured. Furthermore, this dynamic allocation of data identifiers to the worker processes using the iterator concept prevents data skew.

4 Performance and Evaluation

In order to evaluate the speed-up of the parallel RasDaMan server, we chose typical RasQL queries on 2-dimensional and 3-dimensional data. We will first give a detailed example of a test run, describing the test data, the RasQL query, the internal query execution and the intermediate results. Following the running example, parameters influencing the speed-up will be discussed in detail. Finally, the implemented techniques will be compared to techniques used for relational DBMS.

4.1 Running Example

Our example test scenario runs on a collection containing 60 3D MDD. The data was provided by the Max-Planck Institute for Meteorology (partner in the ESTEDI project). Each 3D cube represents temperature values in a specific height above sea level in degrees Kelvin for 10 years (120 months) and was calculated in meteorological simulations (see Fig. 2, left). The spatial domain of a MDD is [0:63,0:127,0:119], the base type is float; the size of one single MDD is 4 MB, the size of the entire collection is 240 MB.

The tests were performed on a SUN Ultra 250 with 2 UltraSPARC II CPUs running at 400MHz, approximately 1.1 GB of main memory and 100 GB of hard disc capacity. The relational DBMS used was Oracle 8.1.6, the parallel RasDaMan server version was 3.5. The query to be analyzed is

```
SELECT all_cells(a > 200.0)
FROM mpim3d as a
```

This simple RasQL statement analyzes each MDD of the `mpim3d` collection and returns a boolean value for each MDD, indicating if all cell values of the MDD are greater than 200.0 degrees Kelvin. This query is particularly well suited for parallel execution because

1. the application includes an induced operation '>' and an aggregation operation 'all_cells' in the operation tree, which are both CPU-bound,
2. the collection includes 60 MDD. The data volume is sufficient to let computational costs dominate over communication costs and to prevent data skew (as shown above, the load is distributed dynamically),
3. the results which have to be collected by the master server process and transmitted to the client are extremely small in this case, i.e. the intermediate result transferred from the worker processes to the master process is only one scalar value per MDD.

For these reasons, this query achieves a speed-up of about 1.91, measured on the RasDaMan server[4]. Typical speed-ups of CPU-bound queries lie within a range of 1.60 and 1.95 (if the response time of the query is not too short, see below). The time required by the different internal process classes is as follows:

- about 2% for the tuple server. This process does no computational work but central administration of the multidimensional objects.

[4] Response times on the client can differ because of slow network transfer rates.

- about 4% for the master process. In this case the result to be transmitted to the client is very small.
- 94% for the two worker processes. As the overall execution time is consumed by the processing of 60 data objects, the data skew in this example is minimal.

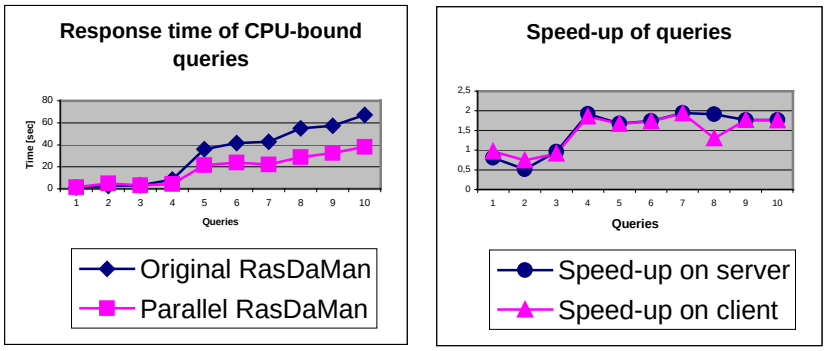

Fig. 3. Performance of the original server compared to the parallel implementation

Fig. 3 shows the performance for 10 CPU-bound queries with ascending response time. We can identify a threshold in the range of 3 seconds. This threshold distinguishes complex queries for which the parallel execution improves performance from simple queries where the overhead of parallelizing exceeds the performance improvement of a parallel execution. However, a response time less than 3 seconds is not typical for the processing array data, so we accept the performance loss for such simple queries. The more complex a query gets, the more speed-up is achieved. Finally, the speed-up of queries having a response time of 30 seconds or more (on the parallel RasDaMan server) is determined only by (1) the kind of array operations (I/O vs. CPU-bound, see section 2.1), (2) the kind of intermediate results which are transmitted by internal processes, (3) the number of MDD which have to be handled. A small number of MDD can lead to data skew.

The speed-up measured on the client application is typically less than the speed-up on the server (Fig. 3, right), especially if the result of the query which has to be transmitted over the network is very large (query 8).

4.2 Evaluation

Intra-query parallelism is a well established technique in relational DBMS, but to our knowledge it has not yet been examined for multidimensional array DBMS. The special properties of array data, such as the enormous size to be processed, the typical size of a single data element (MDD), typical queries being CPU-bound etc., require a detailed evaluation of the parallel algorithms used in relational database technology regarding their suitability for array DBMS:

1. data parallelism vs. pipeline parallelism
   Data parallelism is an excellent way to speed up queries on MDD. Although multidimensional objects typically have a size of several MB (and more) and the opera-

tions are CPU-bound, the communication costs for the distribution of the objects are small compared to the costs for the array operations. Additionally, we tried to minimize communication costs by avoiding to transmit large intermediate results. Pipeline parallelism, on the other hand, destroys the utilization of transient multidimensional objects, i.e. cached MDD, and therefore compromises performance.

2. load balancing and data skew
   Expensive operations on large array data allow for dynamic load balancing. In contrast to relational DBMS which often access millions of very small data tuple array queries typically access up to several hundred data objects having a size of several MB or more. As a consequence, the next MDD to be evaluated can be allocated on demand instead of distributing the workload statically as done in relational systems. This procedure avoids data skew.
3. intermediate results
   Transferring intermediate results proves to be more complex in the case of array data than it is for relational data. Relational database pages filled with tuples have less complexity compared to multidimensional arrays, which consist of complex data of dynamic size, cell type etc. The parallelization module was adapted to this characteristic. Transmission of complex transient intermediate results was avoided where possible.

Summarizing, the implemented parallelism for array data requires methods different from relational techniques, but is well suited for the typical application scenarios involving processing of array data and achieves very good performance.

5 Conclusions

Parallel processing of multidimensional array data in an array DBMS has not attracted any attention in database research so far, although query execution time for processing this kind of data is mostly determined by CPU resources. Our goal was the utilization of parallel hardware in order to speed up CPU-bound queries, especially very expensive queries with execution times of several minutes or even hours. We designed a concept to dynamically split up the computational work on multidimensional objects between multiple processes. This required an adaptation and segmentation of the query tree to allow different parts of the query tree to be executed by different processes. In order to achieve good speed-up we minimized process initialization time and inter-process communication.

The concept described was fully implemented in the RasDaMan server kernel. Extensive test scenarios were performed regarding the structure of the resulting query tree and the intermediate results that have to be transmitted.

Performance measurements prove the validity of our concept. On a two processor machine we observed an increase in speed by a factor of up to 1.91 which is an extremely good result. Further performance measurements on computers with more processors and workstation clusters will follow. We expect similar performance improvements on these architectures as the concept implemented makes no assumptions regarding the number of processes or cluster nodes.

The implemented data parallelism partitions the data with a granularity of entire multidimensional objects. This has the benefit that the concept is straightforward and avoids excessive communication overhead which would lead to a loss of performance. Summarizing, the parallel RasDaMan server shows extremely good performance, especially for computationally expensive queries.

Future work includes the investigation and the implementation of intra-object parallelism. In order to achieve performance improvements for a query which executes array operations on a single MDD, the concept described here is not suitable, as the granularity of our data parallelism is a complete multidimensional object. Speeding up such queries requires splitting up the MDD and processing the resulting fragments in parallel, which has to be investigated further in the future.

References

1. Agrawal, R., Gupta, A. Sarawagi, S.: Modeling Multi-dimensional Databases. Research Report, IBM Almaden Research Center, San Jose, USA, 1995
2. Baumann, P., Furtado, P., Ritsch, R., Widmann, N.: Geo/Environmental and Medical Data Management in the RasDaMan System. In Proc. of the Int. Conf. on Very Large Data Bases (VLDB), Athens, Greece, 1997
3. Baumann, P.: A Database Array Algebra for Spatio-Temporal Data and Beyond. In Proc. of the 4th International Workshop on Next Generation Information Technologies and Systems (NGITS), Zikhron Yaakov, Israel, 1999
4. Bouganim, L., Florescu, D., Valduriez, P.: Dynamic Load Balancing in Hierarchical Parallel Database Systems. In Proc. of the Int. Conf. on Very Large Data Bases (VLDB), Mumbai (Bombay), India, 1996
5. Dehmel, A., Baumann, P.: Visualizing Multidimensional Raster Data with rView. In Proc. of the Int. Workshop on Database and Expert System Application (DEXA), Greenwich, UK, 2000
6. DeWitt, D.J., Gray, J.: Parallel Database Systems: The Future of High Performance Database Systems, Communication of the ACM, Volume 35, 1992
7. DeWitt, D.J., Kabra, N., Luo, J., Patel, J., Yu, J.: Client Server Paradise. In Proc. of the Int. Conf. on Very Large Data Bases (VLDB), Santiago, Chile, 1994
8. Nippl, C., Mitschang, B.: TOPAZ: a Cost-Based, Rule-Driven, Multi-Phase Parallelizer. In Proc. of the Int. Conf. on Very Large Data Bases (VLDB), p. 251-262, 1998
9. Rahm, E.: Dynamic Load Balancing in Parallel Database Systems. In Proc. of EURO-PAR, Lyon, Springer-Verlag, Lecture Notes in Computer Science 1123, S.37-52, 1996
10. Ritsch, R.: Optimization and Evaluation of Array Queries in Database Management Systems, PhD Thesis, Technical University Munich, 1999
11. Tamer Özsu, M., Valduriez, P.: Principles of Distributed Database Systems, Second Edition. Prentice-Hall 1999