Parallel SQL Execution in Oracle 10g

Thierry Cruanes Benoit Dageville Bhaskar Ghosh
Oracle +Corp.
500 Oracle Parkway
Redwood Shores, CA 94065, U.S.A
{thierry.cruanes,benoit.dageville,
bhaskar.ghosh }@oracle.com

ABSTRACT

This paper describes the new architecture and optimizations for parallel SQL execution in the Oracle10g database. Based on the fundamental shared-disk architecture underpinning Oracle's parallel SQL execution engine since Oracle7, we show in this paper how Oracle's engine responds to the challenges of performing in new grid-computing environments. This is made possible by using advanced optimization techniques, which enable Oracle to exploit data and system architecture dynamically without being constrained by them. We show how we have evolved and re-architected our engine in Oracle10g to make it more efficient and manageable by using a single global parallel plan model.

1. Introduction and Overview

Parallel processing is a key technology that allows businesses to efficiently process massive amounts of data. The cost-effective consolidation of hardware resources in a large cluster of interconnected servers - a grid with dynamic allocation and de-allocation of resources - and the support of increasingly complex applications presents new challenges to parallel database systems.

A modern engine needs to be flexible, scalable, resource-efficient and manageable. It should be flexible to adapt to the grid's changing data and system resources without requiring the SQL application to change. The parallel engine of Oracle10g, introduced in Oracle7 [4], is built on top of a shared-disk architecture where every node can access all data. The shared-disk model affords enormous application simplicity because data does not have to be statically partitioned and mapped. It also allows Oracle server to map parallel processing agents to data fragments in a flexible and dynamic manner. Our Parallel Execution (PX) engine is independent of both the underlying hardware architecture and data partitioning, and can therefore exploit the characteristics of the dynamic grid effectively.

Oracle's PX engine achieves scalability by generating efficient parallel plans, guaranteeing balanced workload distribution across parallel agents and minimizing communication costs by a variety of compile and run-time techniques. We optimize the usage critical resources like memory and DML locks by using execution-constructs, which constrain load and thread-allocation in a cost-based manner.

One of the main challenges customers faced with Oracle's PX engine in the past has been the complexity of managing, monitoring, and diagnosing performance of parallel operations. This was historically related to how function-shipping and sub-task representation for parallel execution was implemented. In the Oracle 10g server we solved this by introducing the Parallel Single Cursor (PSC) model where the same global parallel plan for a SQL statement (cursor) is shared across all agents participating in the parallel execution.

The rest of the paper is organized as follows. In Section 2, we provide the basic concepts of Oracle's PX architecture. In Section 3 we summarize the compilation aspects of the PSC model. In Section 4 we show some execution techniques, which make our model cluster-aware, resource-aware and able to dynamically map and allocate resources in a grid. In Section 5 we present some key features of our engine that either parallelize features from different business areas or solve traditionally hard scalability problems. We conclude with some practical lessons specific to our industrial development effort in Section 6. We briefly refer to related work in parallel databases throughout our paper.

2. Concepts and key abstractions

Oracle's PX architecture supports both intra and inter operation parallelism. It is comprised of a Parallel Execution Coordinator (PEC) and a set of Parallel Execution Servers (PES). Physically, a PES is either a thread (e.g. Windows NT) or a process (e.g. UNIX). A set of PESs allocated to execute a parallel statement can run within a node (intra-node parallelism) or across multiple nodes of the grid (inter-node parallelism). Intra-node communication uses shared-memory and inter-node communication uses IPC protocol over high-speed interconnects. Oracle supports parallel execution of a very large spectrum of operations: all relational operations (e.g. scans, joins, order-by, aggregation, set operations), DML's (e.g. insert, update, delete, merge), DDL's (e.g. create table, index and materialized views), data reorganization (e.g. partition maintenance operations), data load and unload via external tables and SQL-extensions for Analytics (e.g. window functions, model clause [12]) and Data-Mining (e.g. frequent item set counting [9])

A parallel execution plan (PEP) is based on four key abstractions: Dataflow operators (DFOs) which are fragments of the parallel execution plan consisting of multiple connected row sources (termed Iterators in [8]), Table Queues (TQs) which encapsulate data redistribution between DFOs (termed Exchange in [8]),

SIGMOD 2004, June 13–18, 2004, Paris, France.

Granule Iterators (GRA) which control horizontal dynamic partitioning of object access and finally the Parallelizer row source which controls the scheduling of a parallel subtree below it. Each such parallel subtree is composed of one or more DFOs each of which can be executed concurrently by multiple PES. In Oracle, this tree-based representation of row sources represents the compiled plan of a SQL cursor.

3. Parallel Single Cursor Model

In the Parallel Single Cursor (PSC) model, all constructs needed for the parallel execution of a statement are expressed as row sources assembled in a single and global execution plan. This execution plan is produced by the PEC when the statement is first compiled and then used by both PEC and PESs when the statement is executed in parallel.

3.1 Generation of a Single PEP

During the compilation of the SQL statement, the optimizer generates a logical execution plan in two passes. In the first pass, join order and access methods are selected such that the parallel cost of the plan is minimized given its degree of parallelism (DOP). While computing the parallel cost of a plan in this pass we take various parallel characteristics into account, such as number of nodes in the grid, number of partitions (table or index fragment) of each object, and the default distribution method. In the second pass, the optimizer computes the best distribution method for each operation in the parallel plan. The result is a logical representation of the execution plan, which then goes through the physical compilation phase to generate a tree-based representation of physical operators (i.e. row-sources). In this phase we also clump row sources into DFO's and perform some physical optimizations by taking into account specific knowledge of the requirement of each row source such as clustering, ordering or data-fragmentation. Unlike the clumping pass of TOPAZ [6] we do not consider the DOP in this phase. Our usage of a parallel cost-model and parallel-aware optimizer is similar to approaches in shared-nothing engines like DB2-PE [1] and Teradata [2].

Figure 1 shows the complete physical plan corresponding to a simple parallel hash-join with its left input serial (e.g. because it is too small to benefit from parallelism). The shaded areas represent DFO's and the white ellipses are row sources. The same physical plan is used by both PEC and PESs. The PEC executes the Parallelizer row source and the serial left DFO (a serial scan). One set of PESs executes DFO1 (a parallel scan) and a second set of PESs executes DFO2 (the parallel hash-join). The PX Send and Receive row sources are added at DFO boundaries to implement the hash redistribution TQ between the two sets of PESs.

3.2 Implication of a Single Global PEP

Most of the industrial parallel database engines express DFOs (i.e. parallel subplans) in some intermediate language (e.g. encoded plan trees that can be transported over a network) or as higher-level SQL constructs. Each approach has benefits, which we cannot elaborate here, for lack of space. However, in our experience with maintaining a vastly complex commercial parallel engine over several releases, we have learned that both approaches have inherent flaws, which become more and more difficult to overcome as the parallel engine grows in size and scope. Generating, compiling and executing an intermediate language for each DFO is a complex and error-prone process. If

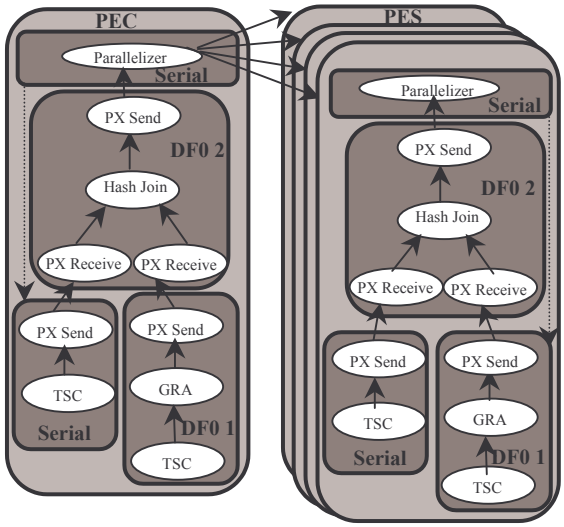

Figure 1. PEP execution on the QC and parallel slaves.

SQL is used as the intermediate representation, generating SQL for each DFO is hard because of the mismatch between the declarative SQL representation and the execution plan based on low-level physical operators. If the intermediate language is a linearized representation of the plan, there is a maintenance issue in making sure that each of the physical operators contains all the information necessary for their execution and in extending the operators every time a new feature is parallelized. To get around such problems, in Oracle10g we avoid using an intermediate language for parallel DFOs altogether. Instead the PEC and all PESs use the same parallel execution plan. This is achieved by sharing the same physical copy of the plan when the PEC and PESs run on the same node, and by shipping the original SQL statement to remote nodes participating in the execution. On each remote node, the statement will be recompiled by one of the PES and shared by the others. This is the Parallel Single Cursor (PSC) execution model. It has three major advantages:

- First, it dramatically improves the manageability and monitoring of parallel execution since all parallel constructs are expressed in the physical plan as row sources. Interesting parallel execution statistics (e.g. memory usage, number of messages, load skew) are all available in the row sources of the shared plan across all PESs and can be aggregated either live (for progress monitoring) or after a query finishes.
- Second, parallelization of new operation becomes much easier and less error-prone since we do not have to support an intermediate language to implement function shipping. The only additional step needed to parallelize a new operation is to define data distribution, clumping and DFO boundaries for the corresponding new row sources.
- Third, it improves code maintenance and debugging capability and reduces the memory usage associated with intermediate representations.

Our decision of using a shared-plan model but foregoing an intermediate representation for function-shipping stands in contrast to Informix XPS [13].

4. Execution of a Parallel Single Cursor

As discussed above, the PEC and all PESs use and even share the same parallel plan. At execution time, the PEC only sends control information to PESs to schedule the various DFOs (defined by the dotted arrows in Figure 1). PESs scanning an object, lazily request and receive object fragment information (a granule) via the granule row source (GRA). Once a PES has finished consuming a granule, it requests another one from the PEC. Hence, dynamic load balancing happens automatically by the rate at which PESs consume granules and generate data to be sent to the parent DFO.

One of the main advantages of our PX engine is that it is neither constrained by the physical layout of the database (the hardware configuration) nor the logical layout of the database (how the database objects are partitioned by the user). It takes advantage of the physical layout by imposing vertical virtual partitioning of the dataflow for various optimizations, some of which are traditionally found in shared-nothing architectures [1] and some of which are only possible because of our shared-disk approach.

With shared-nothing systems, the database must be partitioned in advance so that a given node can access a fragment of the data (e.g. by HASH in Teradata [2]). The scheduler has to be aware of this static node-to-data mapping and cannot use any other node to access the data. No such constraints burden Oracle's PX engine. In fact, we leverage Oracle's Partitioning Option which supports data-fragmentation for high-availability and performance. Our PX engine is not constrained by how users use partitioning to fragment database objects (which should be based on business requirements), but can exploit it whenever possible.

For example, we exploit user-defined partitioning to implement our version of collocated equi-joins[1], even though strictly there is no concept of location in our model. In this version, which we term Full Partition-wise Join, each PES is constrained to scan and join an entire partition from each of two "equi-partitioned" tables. However, unlike in shared-nothing architectures, any PES can perform the partition-wise join for any of the partition pair since all nodes share disks. For example, the scan-join for a particular partition pair can be performed by a PES running on node 1 in the first run of a query, and by a different PES running on node 10 in the next execution of that query.

4.1 Cluster-aware PX

During the optimization phases of a parallel operation, the optimizer considers the number of nodes available and the number of data partitions involved in the query, and generates a vertical partitioning of the dataflow to minimize communication costs.

For instance, consider a join between two equi-partitioned tables performed on a cluster of two nodes. Both tables are partitioned into 4 partitions. The best way to minimize communication cost is to eliminate the need of distributing the data altogether. If the requested degree of parallelism (DOP) is four, a regular full partition-wise join can be done. The join of a pair of partitions, one from each table, is processed by a single PES and no data redistribution is performed as shown in Figure 2.
However, with partition-wise join, the maximum DOP is limited by the number of partitions. In the above example a maximum of four PESs can be used.

An alternative strategy is the parallelized hash-join algorithm. This strategy can be selected when the overhead in hash-redistributing each row is outweighed by the gain of using more PESs. In our example we could use a DOP of 8 (which means that

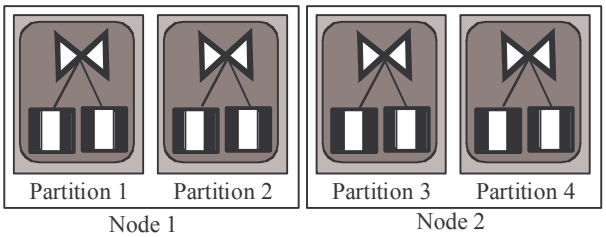

Figure 2. Full Partition-wise Join of 4 partition tables on 2 nodes.

16 PESs are used, 8 producers and 8 consumers), but redistributing both inputs on the join key would be required. Each PES scanning the partitioned table sends its rows to a join PES after computing a hash function. On a cluster, the scan to join traffic can go through the cluster's interconnect (as illustrated in Figure 3) and therefore incur a higher communication cost.

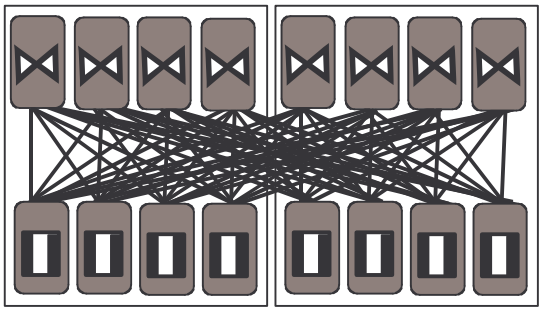

Figure 3. Hash join on 2 nodes with a DOP of 8.

A better execution strategy is a hybrid of the above two solutions. The main idea is to split the PES set according to the physical constraints, leveraging the underlying partitioning scheme for optimizing the redistribution of data. A constraint on the execution of a parallel query is expressed through a special construct called a PES mapper attached to each of the objects. This class of physical optimizations is especially important on grid architectures composed of a high number of small nodes since in this case most of the communication traffic goes through the cluster's interconnect. When executed on multiple instances (cluster or MPP installation) this strategy avoids all data traffic through the cluster's interconnect and keeps communication local to each node. This execution strategy can be chosen by the optimizer in the situation where a partition-wise join is possible but causes the DOP to be reduced because of the small number of partitions. The optimal strategy for a given environment is chosen based on CPU and communication costs of all of the above variants of the execution plan.

Figure 4 shows the same hash-join example with eight PESs using the PES-mapper optimization. In environments with disk affinity to nodes, the assignment of partition to the nodes of the cluster obeys the affinity of the data. In this mode, Oracle behaves like a shared-nothing cluster but is still able to take advantage of shared-disk load-balancing optimizations.

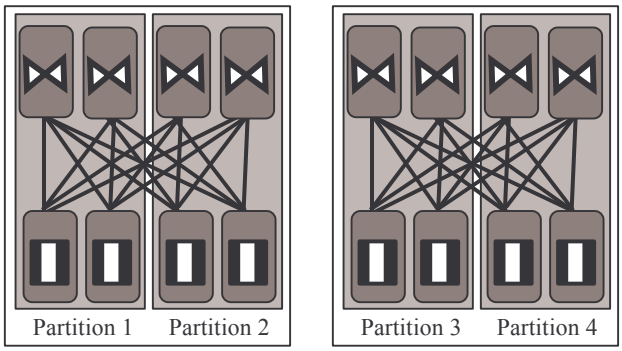

Figure 4. Hash join on 2 nodes with a PES mapper.

4.2 Performance

For reasons of space, we cannot present and analyze performance data in detail. Table 1 shows how cluster-aware PES mappers improve the performance of a simple two-table hash-join (modeled on Query 19 of the TPCH schema). In the example, 600 Gbytes of data is sent over a cluster of 1 to 8 nodes each with four Intel Xeon processors running Linux. With a PES mapper, there is no interconnect traffic except for messages needed to return the result to the PEC. This improves the scalability of this interconnect-bound query. The single node number for elapsed time shows the relatively small overhead of using a PES mapper due to the additional CPU cost of the mapping function applied to each row sent from a scan PES to a join PES.

Table 1. Use of PES Mapping (PM) with Hash distribution

	Without PM	With PM
8 nodes	113s	92s
4 nodes	202s	156s
1 node	606s	617s

4.3 Resource-Aware PX

Parallelism puts a heavy burden on the resources of any system and it is very easy for its resource usage to grow super-linearly and overwhelm the physical limits of any system. Our PX engine transparently adapts the usage of resources like memory, locks and processes to system load.

Memory. The memory requirement for parallel execution can be very large (e.g. parallel hash-join requires each PES to build an in-memory hash-table). In order to control the amount of memory consumed at runtime, each SQL operator dynamically adapts to the memory pressure on the system by communicating with the memory manager [3]. In some cases it is possible to limit the memory consumption of some operators a priori. For example, in a parallel INSERT statement against a partitioned table, each PES allocates in-memory buffers for each partition it loads into. These in-memory buffers are used to directly perform large and asynchronous write for the rows being loaded. If the memory available is not sufficient, it is necessary to limit the number of partitions touched by a given PES. In this case the PES mapper construct splits the PESs into small enough groups working on distinct partitions, thus preventing the DML from running out of memory since a given PES will work on a subset of partitions.

Locking. A parallel DELETE or UPDATE statement can easily overcome the physical limit on the number of processes allowed by the Oracle lock manager to modify a single data block. In order for the DML operation to succeed, the engine has to control the number of PESs touching a block. In this case, the PES mapper splits PESs into groups similar to the memory-optimization above, to respect the lock limit.

Adaptive DOP and PES Allocation. In the grid, each node has information about the current workload, the number of CPU's, and the virtual service supported. We use this information at the beginning of a parallel query to constrain the DOP. A user connects through a session associated with a service (e.g. finance or marketing). When this session starts a parallel query, the PEC tries to satisfy the query requirements by acquiring PESs only on nodes supporting the service. Given a set of nodes, the load-balancing algorithm tries to maintain an even load across the nodes and to minimize the inter-node communication cost. A unit of allocation per PES is computed based on the workload and the DOP requested. This unit of allocation is used to fill up each of the nodes until they reached their load target.

Granule allocation. During allocation of granules (unit of work or fragment of objects) the data-locality or affinity (of a disk to a node) can be taken into account by the PEC to map the granules to PESs. The result of this optimization pass is a set of work queues from which the PESs get their granules first. Because of the shared-disk model, this optimization problem can be solved without rebuilding the database (i.e. re-defining the layout of the data) and by dynamically reconfiguring resource usage by taking into account the shape of the service offered by the grid.

5. A feature-rich Parallel Engine

A modern engine has to go beyond the support of the simple parallel operators such as scan, join and aggregation. We now illustrate how different properties of Oracle's PX engine are used to transparently parallelize an ETL process, to parallelize traditionally problematic queries such as correlated subqueries, and to support requirements coming from new business areas such as the integration of data-mining algorithms in the parallel engine.

5.1 Flexible data distribution and pipelining

The key element in SQL parallelization of functions and operators is the partitioning of input data objects (e.g. tables or indexes) or streams (intermediate results in a query execution plan). Oracle supports the simple and natural parallelization of a large class of features by using the explicit or implicit distribution requirements of the physical operator.

The optimizer can infer the distribution from a suitable grouping construct (e.g. PARTITION BY for an OLAP Window Function, PARTITION BY and DIMENSION BY for SQL MODEL clause[12]) to parallelize features transparently. Consider a parallel operation aggregating the sales revenue per region followed by an order-by on region. Oracle would choose a hash-based redistribution on region id between the scan and sort PESs as if the order-by did not exist. In this case the clumping logic during physical plan generation would recognize the compatibility between the group-by and order-by keys and would change the hash redistribution to range to therefore eliminate the subsequent DFO with the order-by sort.

For a user-defined or explicit specification of data redistribution, consider a user-defined Table Function [9] with a PARTITION BY HASH clause on column a. The query "*SELECT a', b' FROM TABLE(F(CURSOR(select a, b from TAB))*" would be compiled into a parallel scan of TAB, followed by a redistribution of the rows by hashing on column a to feed the parallel evaluation of F across the PESs.

To illustrate the power of this mechanism, consider an ETL (Extract-Transform-Load) process by which data is read from external sources, cleaned and transformed through user-defined Table Functions, and then loaded (via MERGE or multi-table INSERT) into one or more tables. In Oracle 10g, this entire process is performed inside the database server through a single SQL statement. Oracle transparently parallelizes this statement without any intermediate synchronization or staging. The external table is scanned in parallel by dividing the external object into load-balanced granules. The output rows are fed into a parallel table function obeying the PARTITION BY clause specified in its definition and the resulting rows are re-partitioned according to the requirement of the destination table amongst the PESs used for the DML operation. If the partitioning requirements of the DML and the Table Function are compatible, then the two operations are clumped together, avoiding data redistribution.

5.2 Cost-based Parallelization of Subqueries

Parallelization of subqueries is a difficult problem which most of commercial engines handle either through subquery flattening techniques (e.g. Teradata [2], magic decorrelation in DB/2[10], transformation via aggregation in SQL server[5]) or simply going serial in Sybase [11]). Oracle supports most of the query rewrite techniques transforming nested subqueries into joins and aggregations, but for queries that cannot be unnested or transformed we parallelize the query plans in a general cost-based manner. The benefit of the PSC model is that we can build independent parallel plans for nested query blocks and decide at run-time which query block should go serial either because it is inside a PES or because the overhead of restarting a parallel subquery plan for each input row is too high.

5.3 Recursive and Iterative Computation

Oracle supports parallel recursive and iterative computation by using temporary tables between DFOs. For instance, the Frequent Itemset (FIS) Counting algorithm is natively supported by our parallel engine and is used as the core computational routine by Oracle Data-Mining to compute association rules. The core of the FIS algorithm is to iteratively compute the most frequent k+1 items from the most frequent k items. This is mapped onto two DFOs by computing the k item-set on DFO 1, repartitioning the data rows to DFO 2, which computes the k+1 item-set, inserts the results in parallel into a temporary table and then repartitioned and distributed amongst PESs executing DFO 1 which switches phases to compute the k+2 item-set. Data repartitioning happens via the regular redistribution mechanism in one phase (DFO 1 to 2) and via dynamic partitioning of the temporary table in the next phase (DFO 2 to 1). This general construct can be extended to support any algorithm in need of a synchronized parallel loop.

5.4 Load-Balanced Table Queues

For data redistribution strategies like range (e.g. used in global order-by, or create parallel index) which are sensitive to the statistical properties of the ordering keys in the input stream, Oracle supports runtime techniques for dynamic sampling which can provide better splitting elements so as to prevent data skew in the consumer PESs.

6. Conclusions

In this paper we have focused on three main aspects of the Oracle 10g parallel execution engine.

First, we have presented the redesign and evolution of the Oracle's parallel SQL execution architecture to eliminate the need for an intermediate language for function-shipping. Instead we use a new parallel single-plan based model, which keeps our internal architecture clean, maintainable and extensible as parallel execution becomes more ubiquitous and transparent for the backend. Second, we have stressed how our execution model is completely decoupled from data-placement. This allows us to enable novel optimizations like PES-mapping which can be used to optimize a variety of metrics like distribution cost, memory usage and lock contention. We have found that the shared-disk model of parallel SQL execution has afforded us flexibility to scale in new and ever-evolving hardware environments. This experience runs contrary to much of conventional wisdom in literature on parallel databases and we hope this helps future practitioners in this area. Third, we have presented some of the features and capabilities of our PX engine that help solve either traditionally hard problems (e.g. DML, subqueries) or allow new functionalities like ETL and Data-Mining to be moved from tools into the backend.

7. References

[1] C.K.Baru, G. Fecteau et. al.: DB2 Parallel Edition. IBM Systems Journal, Vol 34, No 2, 1995.

[2] C. Ballinger and R. Fryer: Born to be Parallel. Data Engineering, 20(2). June 1997.

[3] B. Dageville, M. Zaït: SQL Memory Management in Oracle9i. In: Procs of 28th VLDB Conference, 2002.

[4] G. Hallmark Oracle Parallel Warehouse Server. ICDE 1997.

[5] C. Galindo-Legaria, M. Joshi: Orthogonal Optimization of Subqueries and Aggregation. ACM SIGMOD, 2001.

[6] C. Nippl, B. Mitschang: TOPAZ: a Cost-Based, Rule-Driven, Multi-Phase Parallelizer. 24th VLDB. 1998.

[7] G. Graefe: Volcano - An Extensible and Parallel Query Execution System. TKDE, 6(1), 1994.

[8] G. Graefe: Query Evaluation Techniques for Large Databases. ACM Computing Surveys 25(2), June 1993.

[9] Oracle DBMS 9.2 Data Warehousing Guide, 2002

[10] P. Seshadri, H. Pirahesh, and T. Y. C. Leung. Complex query decorrelation. ICDE 1996.

[11] Sybase Adaptive Server Enterprise, Performance and Tuning, Optimizer and Abstract Plans, 2003.

[12] A. Witkowski et. al.: Spreadsheets in RDBMS for OLAP. ACM SIGMOD Conference 2003 : 52-63.

[13] C. Zhou: XPS: A High Performance Parallel Database Server. In: Data Engineering, 20(2). June 1997.