Parallel Query Processing in Shared Disk Database Systems

Erhard Rahm

University of Kaiserslautern, Germany
E-mail: rahm@informatik.uni-kl.de

Abstract: System developments and research on parallel query processing have concentrated either on "Shared Everything" or "Shared Nothing" architectures so far. While there are several commercial DBMS based on the "Shared Disk" alternative, this architecture has received very little attention with respect to parallel query processing. A comparison between Shared Disk and Shared Nothing reveals many potential benefits for Shared Disk with respect to parallel query processing. In particular, Shared Disk supports more flexible control over the communication overhead for intra-transaction parallelism, and a higher potential for dynamic load balancing and efficient processing of mixed OLTP/query workloads. We also sketch necessary extensions for transaction management (concurrency/coherency control, logging/recovery) to support intra-transaction parallelism in the Shared Disk environment.

1 Introduction

Parallel database systems are the key to high performance transaction and database processing [DG92, Va93a]. These systems utilize the capacity of multiple locally distributed processing nodes interconnected by a high-speed network. Typically, fast and inexpensive microprocessors are used as processors to achieve high cost-effectiveness compared to mainframe-based configurations. Parallel database systems aim at providing both high throughput for on-line transaction processing (OLTP) as well as short response times for complex ad-hoc queries. This requires both inter- as well as intra-transaction parallelism. Inter-transaction parallelism (multi-user mode) is required to achieve high OLTP throughput and sufficient cost-effectiveness. Intra-transaction parallelism is a prerequisite for reducing the response time of complex and data-intensive transactions (queries).

So far, the use of intra-transaction parallelism has mainly been studied for "Shared Everything" (SE) or "Shared Nothing" (SN) architectures. While several commercial DBMS support the "Shared Disk" (SD) alternative (IMS, Rdb, Oracle, Ingres, AIM, etc.), these systems are currently restricted to inter-transaction parallelism[1]. The current world record in the TPC-B benchmark [Gr91] is held by Oracle's SD system called "Parallel Server". In 1991, more than 1000 tpsB at a remarkably low cost were achieved on a Ncube system with 64 nodes. Given that more and more microprocessor-based "cluster" architectures support the shared-disk paradigm (Sequent, Pyramid, Encore, etc.), we expect a growing significance of the SD approach for parallel database processing.

Despite the significance of SD for high performance database processing, this approach has found almost no attention in the open research literature with respect to intra-transaction parallelism. Since many researchers consider SN as the major architecture for parallel query processing, we discuss the SD approach by comparing it with SN. For this purpose, we first compare some general features of both architectures with respect to database processing (section 2). This discussion reconsiders some of the arguments that have been made to promote SN as "the" approach for parallel query processing. In section 3, we extend our comparison by focussing on parallel query processing for both architectures. The comparison is not intended to show that SD is "better" than SN, but to illustrate that there are major advantages for SD which make this approach an interesting target area for further research on parallel query processing. In particular, major problems of the SN approach with respect to intra-transaction parallelism (e.g., physical database design, support for mixed workloads) are likely to be easier solved for SD[2]. In section 4, we discuss extensions for transaction management that are to be supported by SD systems for intra-transaction parallelism.

2 SN vs. SD revisited

We assume familiarity with the basic differences between SN and SD. SN systems are based on a physical partitioning of the database among processing nodes, while in SD systems each node has access to all external storage devices and thus to the complete physical database. Transaction/query execution is distributed for SN if access to multiple database partitions is needed; communication is also required for the commit protocol. SD requires inter-node communication for global concurrency control and coherency control [Ra91, Ra93a].

The key problem of SN is finding a "good" fragmentation and allocation of the database. The database allocation has a profound impact on performance since it largely determines where database operations have to be processed thus affecting both communication overhead and node utilization. For

[1] Oracle is currently working on intra-query parallelism [Li93].

[2] Given the problems of SN, Valduriez has also recently advocated for a "shared-something" approach, that is a SD system in which each node is itself a multiprocessor [Va93b].

complex queries, the database allocation must also support an effective intra-transaction parallelism. Since different transaction and query types have different data distribution requirements, the database allocation must inevitably constitute a compromise for an expected workload profile. The chosen allocation may however easily lead to suboptimal performance and load imbalances due to short-term workload fluctuations or other deviations in the actual from the expected workload profile. Variations in the number of nodes (node failure, addition of new node) require a reallocation of the database. SD avoids these problems since there is no need to physically partition the database among nodes. In particular, this promises a higher potential for load balancing since each node can process any database operation. We will further discuss the role of the database allocation in the next section.

One advantage of SN is that interconnecting a large number of nodes is less expensive than for SD since every disk needs only be connected to one node. However, the cost of interconnecting nodes and disks is also comparatively low in SD architectures such as Ncube. These architectures are based on microprocessors and do not directly attach a disk drive to all nodes, but achieve the same connectivity by an interconnection network (e.g., hybercube) where I/O requests and pages may be transferred through intermediate nodes. Such a message-based I/O interface between processing nodes and I/O nodes (disk controllers) is also used in the DEC Vax-Clusters. This approach does not imply any inherent limit on the number of processing nodes.

In [DG92], it is speculated that coherency control may limit the scalability of SD compared to SN. However, for workloads for which SN systems could demonstrate scalability so far, namely read-dominated workloads or perfectly "partitionable" loads like debit-credit (TPC-A, TPC-B), SD has no problems with coherency control. It is fair to say that SN systems depend even more than SD on the partitionability of the workload and database due to the large performance impact of the database allocation for SN [YD91]. SD can utilize the partitionability of a workload to limit the number of buffer invalidations by employing an "affinity-based" workload allocation which assigns transactions referencing/updating the same database portions to the same node [Ra92]. This is a dynamic form of workload/data allocation which can more easily be adapted than the database allocation for SN. Furthermore, SD allows very efficient coherency control by closely integrating this function into the lock protocol [Ra91].

In [DG92] it is also argued that SN would scale better than SD because interference is minimized. In particular, SN would only move small "questions" and (filtered) answers through the network, while SD would require moving large quantities of data. However, this argument no longer holds for parallel query processing for which SN also requires redistribution of large amounts of data, e.g. for join processing (section 3). Furthermore, SN is highly susceptible to processor interference since questions on a particular database object can only be processed by the owning node, even if it is already overutilized. The performance results published for SN so far were typically based on many best-case conditions regarding workload profile, database allocation and load balancing. In particular, almost all performance studies on the use of intra-transaction parallelism assumed single-user mode implying minimal processor interference. Under more realistic conditions (multi-user mode, mixed workloads), SN was shown to exhibit much poorer performance [MR92].

3 SN vs. SD for parallel query processing

The comparisons of the different architectures made so far in the literature did not consider intra-transaction parallelism in most cases. Even in papers coping with parallel database processing [Pi90, DG92, Va93a], no special attention was paid to parallel query processing for SD. In this section, we show that SD offers significant advantages for parallel query processing compared to SN. The comparison considers differences and commonalities with respect to various parallelization forms, database allocation and processing of scan and join operations. Furthermore, we discuss processing of mixed OLTP/query workloads.

3.1 Types of intra-query parallelism

Several forms of intra-query parallelism can be distinguished, particularly intra- vs. inter-operator parallelism and pipeline vs. data parallelism [DG92]. The differences between SN and SD with respect to inter-operator and pipeline parallelism are comparatively small. This is because they primarily work on derived data that can dynamically be redistributed among nodes for both architectures. Hence, our analysis concentrates on data parallelism and intra-operator parallelism.

Data parallelism requires both I/O parallelism and processing parallelism. *I/O parallelism* means that the data to be processed by a database operation is declustered across multiple disks so that I/O time is not limited by the low bandwidth of a single disk. *Processing parallelism* requires that the input data of a database operation can be processed by multiple CPUs to avoid that execution time is limited by the capacity of a single processor. For SN and SD, the database allocation to disk directly determines the maximal degree of I/O parallelism per relation. Since the data allocation on disk is expensive to change, the (maximal) degree of I/O parallelism is a rather static parameter. For SN, the degree of processing parallelism is also largely dependent on the static data allocation to disk since each disk is exclusively assigned to one processing node. This results in a reduced flexibility for dynamically varying the degree of processing parallelism compared to SD where each node can access any disk.

3.2 Database allocation

Declustering in SN systems is typically based on a horizontal fragmentation and allocation of relations. Fragmentation may be defined by a simple round robin scheme or, more commonly, by hash or range partitioning on a *partitioning attribute* [DG92]. Data allocation incorporates determination of the *degree of declustering* D and mapping of the fragments to D disks (processing nodes). Determination of an appropriate database allocation means finding a compromise with respect to contradicting subgoals: support for a high degree of intra-transaction parallelism, low communication overhead, and effective load balancing. For instance, a high degree of declustering supports intra-transaction parallelism and load balancing, but at the expense of a high communication overhead for starting and terminating suboperations. A small degree of declustering, on the other hand, reduces communication overhead and may be sufficient to meet the response time requirements on small relations or for selective queries (e.g., index scan). Furthermore, it supports effective inter-transaction parallelism (high OLTP throughput).

For SD, only a database allocation to disk needs to be determined as already for centralized DBMS. The declustering of relations across multiple disks can be defined similarly as for SN, i.e., either based on round robin, hash or range partitioning. The round robin approach may even be implemented outside the DBMS, e.g., by the operating system's file manager or, in the case of disk arrays [PGK88], by the disk subsystem. (File blocks rather than records would then constitute the units of declustering). In this case, the DBMS optimizer would still have to know the degree of declustering to allocate resources (CPUs, memory) for parallel query processing. Note that centralized DBMS and SN systems typically are unable to utilize the I/O bandwidth provided by disk arrays. This is because disk arrays can deliver I/O bandwidths in excess of 100 MB/s, while it is estimated that a single CPU can process relational data at a rate of merely 0.1 MB/s per MIPS [GHW90]. For SD the CPU bottleneck is avoided if multiple processing nodes share a disk array and if the disk array is able to split the output of a single read request among multiple nodes (memories). Hence, SD is better positioned than SN or SE to utilize disk arrays for parallel query processing.

We note that database partitioning will become more difficult for ***next-generation applications***. For instance, large multi-media objects can be stored in a single tuple ("long field") so that they would be assigned to a single node in SN systems. Hence, parallelism cannot be utilized for SN to process such large objects. For SD, on the other hand, the object could physically be declustered across multiple disks so that at least I/O parallelism could be utilized to reduce I/O time. Similarly, complex objects for engineering applications are typically large and consist of many inter-connected and heterogeneous tuples. Partitioning these objects among multiple nodes is difficult and would introduce a high communication overhead for object processing. Even partitioning at the object level is problematic due to subobjects that are shared by multiple complex objects. Hence, SD is better able to support intra-transaction parallelism on complex-object and object-oriented databases.

3.3 Scan

Scan is the simplest and most common relational operator. If predicate evaluation cannot be supported by an index, a complete *relation scan* is necessary where each tuple of the relation must be read and processed. An *index scan* accesses tuples via an index and restricts processing to a subset of the tuples; in the extreme case, no tuple or only one tuple needs to be accessed (e.g., exact-match query on unique attribute).

For SN, parallelizing a scan operation is straight-forward and determined by the database allocation. For hash and range partitioning, exact-match queries on the partitioning attribute can be restricted to a single processor; range partitioning also allows restricting the number of nodes for range queries on the partitioning attribute. However, all other scan queries must be processed by all nodes holding fragments of the respective relation. This approach has the obvious disadvantage that it does not support dynamic load balancing, i.e., varying the number of processing nodes and selecting the scan nodes according to the current system state. Furthermore, for selective queries supported by an index it is generally inefficient to involve all nodes holding a fragment of the relation due to an unfavorable ratio between communication overhead and useful work per node. The latter disadvantage can be reduced by a multidimensional range partitioning approach [GDQ92]. In this case, fragmentation is defined on multiple partitioning attributes so that queries on each of these attributes can be limited to a subset of the fragments/ nodes. While this approach can reduce communication overhead for certain queries compared to one-dimensional range partitioning, the degree of processing parallelism and thus the communication overhead are still statically determined by the database allocation.

In SD systems each node has access to the entire database on disk. Hence, scan operations on a relation can be performed by any number of nodes. For example, index scans on any attribute may be performed by a single processor thereby minimizing communication overhead. This would especially be appropriate for exact-match and selective range queries, and supports high OLTP throughput. For relation scans, on the other hand, a high degree of processing parallelism can be employed to utilize intra-query parallelism to reduce response time and to achieve load balancing. Not only the degree of processing parallelism can be chosen based on a query's resource requirements, but also which processors should perform the scan operations. Furthermore, both scheduling decisions can be drawn according to the current system state. For instance, a scan may be allocated to a set of processors with low CPU utilization in order to avoid interference with concurrent transactions on other nodes.

The ability to dynamically determine the degree of scan parallelism and the scan processors represents a key advantage of SD compared to SN. It is critical for a successful use of intra-transaction parallelism in multi-user mode where the current load situation is constantly changing. This is because the optimal degree of intra-query parallelism (yielding the best response time) strongly depends on the system state and is generally the lower the higher the system is utilized [RM93].

The result of a scan operation may be too large to be kept in main memory at the query's home node so that it must be stored in a temporary file on disk. In the case of SN, a high overhead is necessary to send the local scan results to the query's home node, to write out the data to disk and to read it in later to perform some postprocessing and return it to the application/user. SD can avoid the communication overhead for the data transfers since each scan processor can directly write its local scan result to a temporary file on the shared disks (or in shared semiconductor memory). After the temporary file is written, the query's home node is informed that the file can be read from external storage.

3.4 Join

Parallel (equi-)join processing typically consists of a parallel scan phase and a parallel join phase. During the scan phase, the scan processors read the input relations from disk and perform selections on them. The scan output is then redistributed among multiple join processors performing the join phase using any sequential algorithm (e.g., hash join or sort-merge). Finally, the local join results are merged at a designated node. Typically, data redistribution between scan and join processors is performed by applying a hash function on the join attribute. This ensures that matching tuples of both input relations arrive at the same join processor.

Even for SN there is a high potential for dynamic load balancing since the join processors obtain their input from the scan processors and do not operate on base relations. In particular, the number of join processors as well as the choice of these processors can be based on the current load situation. Simulation experiments have shown that such a dynamic approach is necessary for efficient parallel join processing in multi-user mode [RM93].

The sketched approach is also applicable to SD. However, there are still significant advantages for SD with respect to parallel join processing. First, SD supports dynamic load balancing not only for the join phase but also for the scan phase. Furthermore, the possibility to exchange intermediate data across shared storage devices can greatly reduce communication overhead for redistributing data between scan and join processors. This is particularly advantageous for large intermediate results which cannot be held memory-resident at the join processors [Ra93b]. Finally, index-supported join queries that only require access to few tuples can be limited to one (or a few) node(s) for SD, while a high communication overhead may be necessary for SN.

A general SN approach to process θ-joins (non-equi joins) between two relations is to dynamically replicate the smaller relation at all nodes holding a fragment of the larger relation and to perform the θ-joins at the latter nodes. This approach causes an enormous communication overhead and does not scale well since the communication overhead increases quadratically with the number of nodes (holding fragments of the two relations). SD avoids the communication overhead for data redistribution altogether since each node can directly read all fragments from disk. Furthermore, the number of join processors is not predetermined by the degree of declustering but can dynamically be selected. A high degree of declustering with respect to the disk allocation is favorable to reduce disk contention for SD. Disk contention is also reduced by the use of large main memory caches and disk caches.

3.5 Mixed workloads

Supporting mixed OLTP/query workloads is already difficult in centralized (or SE) DBMS due to increased resource and data contention. In parallel database systems, there are two additional areas where performance problems for mixed workloads may occur: communication overhead and load balancing. Intra-query parallelism inevitably causes increased communication overhead (compared to a sequential execution on one node), leading to higher resource utilization and contention and therefore lower throughput. To limit the communication overhead and resource contention and to effectively utilize the available processing capacity, dynamic load balancing is particularly important for mixed workloads. As the preceding discussions have already shown, SD offers advantages over SN in both areas:

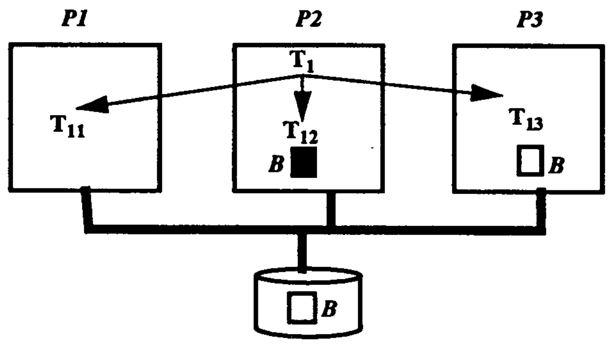
- SN cannot efficiently support both workload types, but requires definition of a (static) database allocation for an "average" transaction profile. This inevitably leads to sub-optimal performance for both workload types and does not support dynamic load balancing. In particular, ad-hoc queries have to be restricted to fewer nodes than desirable to limit the communication overhead so that response times may not sufficiently be reduced. On the other hand, OLTP transactions cannot be confined to a single node in many cases thereby causing extra communication overhead and lowering throughput. In both cases, the sub-optimal performance must be accepted even if only one of the two workload types is temporarily active.
- In SD systems, declustering of data across multiple disks does not increase the communication overhead for OLTP. In general, OLTP transactions are completely executed on one node to avoid the communication overhead for intra-transaction parallelism and distributed commit. The degree of processing parallelism and thus the communication overhead for ad-hoc queries can be adapted to the current load situation. Furthermore, resource contention for CPU and memory between OLTP transactions and complex queries may largely be avoided by assigning these workload types to disjoint sets of processors which is not possible for SN, in general.

4 SD Transaction Management for Parallel Query Processing

Without intra-transaction parallelism, there is no need for distributed transactions for SD. Each node can perform all database operations since the entire database is directly accessible. In particular, all modifications by a transaction are performed at one node and logged in this node's local log file. Hence, no distributed commit protocol is necessary as for SN to guarantee the ACID properties. Communication is necessary for global concurrency and coherency control. Furthermore, the local log files have to be merged into a global log to support media and crash recovery.

However, intra-transaction parallelism results in a decomposition of transactions and queries into multiple subtransactions running on different nodes. In SN systems, the separation of global transactions into subtransactions is determined by the database allocation. This approach typically ensures that each subtransaction operates on data owned by the respective node. Concurrency control is a local function since each node can process all lock requests on its data. For SD, on the other hand, it cannot generally be excluded that subtransactions of a given transaction reference and modify the same database objects (e.g., index pages) at different nodes. Hence, there is a need for concurrency control between parallel subtransactions. Furthermore, coherency control is also required between parallel subtransactions to avoid access to obsolete data and to propagate updated database objects between subtransactions (Fig. 1).

The new requirements can be met by supporting a nested transaction model and by extending the SD concurrency/coherency control schemes for sequential transactions accordingly. Since the applications should remain unchanged compared to sequential transaction processing, nested transactions are only used internally by the DBMS to structure



Subtransactions of transaction T_1 are running in parallel at processing nodes P1, P2 and P3. Subtransaction T_{12} at P2 has modified page B in main memory; the copies of B at P3 and on disk are obsolete. When subtransactions T_{11} and T_{13} want to read page B, a synchronization with

Fig. 1: Concurrency/coherency control problem between subtransactions

queries into a hierarchy of subtransactions or subqueries. Subtransactions can be executed concurrently at different nodes. Furthermore, subtransactions may be rolled back without impact on other subtransactions, i.e., the scope of undo recovery can be substantially limited compared to flat transactions. Isolation between subtransactions is achieved by a suitable locking protocol defining the rules for lock processing within a transaction hierarchy. Such a protocol supporting parallel subtransactions, upward and downward inheritance of locks as well as multiple lock granularities has been proposed in [HR93] and can be extended to the SD environment. One difference for SD results from the fact that lock requests may have to be sent to a global lock manager thus incurring communication overhead and delays. Furthermore, coherency control must be incorporated into the concurrency control scheme. Fortunately, this can be accomplished in a similar way than for sequential transactions, e.g., by a so-called on-request invalidation protocol that uses extended global lock information to detect obsolete page copies and to record where a page was modified most recently [Ra91]. Such an approach detects obsolete page copies during lock processing so that extra messages for this purpose are avoided.

In addition to the extensions needed for concurrency and coherency control, parallel update transactions in SD systems require major changes for logging and recovery. In particular, the updates of a single transaction may now be performed at multiple nodes so that a transaction's log data are spread over multiple local log files. While this is also the case for SN, SN has the advantage that each local log file only contains log data of one database partition thereby supporting media recovery without the need for a global log file. Hence, parallel update transactions for SD would require support of the logging and recovery protocols of both SN (distributed commit, distributed undo and redo recovery) and SD (construction of a global log file). Possibilities to avoid the complexity and overhead of supporting both recovery paradigms are discussed in [Ra93b].

The mentioned problems are largely introduced by parallel update transactions. The implementation complexity for supporting intra-transaction parallelism for SD can thus largely be reduced if merely read-only transactions/queries may be parallelized. In this case, no changes are necessary with respect to logging and recovery compared to conventional SD systems. Furthermore, lock conflicts between concurrent subtransaction of the same transaction are avoided as well as the need for coherency control within transactions. Still, most of the performance gains of intra-transaction parallelism can be obtained since complex transactions/queries are mostly read-only. Executing update (OLTP) transactions sequentially at one processing node also reduces communication overhead thereby supporting high transaction rates.

Even with intra-transaction parallelism be restricted to read-only transactions, a nested transaction model is still needed to propagate locks within the transaction hierarchy (e.g., between main transaction and subtransactions). Furthermore, a hierarchical lock protocol with multiple lock granularities should be supported to keep the concurrency control (communication) overhead for complex queries small. For instance, a relation scan may lock the entire relation with one global lock request. Subtransactions performing scan operations on different relation fragments in parallel do not have to request additional locks, but may inherit a read lock on their relation fragment from the parent transaction. At the end of a subtransaction, the fragment lock is returned to the parent transaction which can finally release the entire relation lock with a single message. Such a hierarchical lock protocol has also to be extended to support coherency control as discussed in [Ra93b].

5 Concluding remarks

Shared Disk (SD) database systems promise significant advantages for parallel query processing compared to Shared Nothing (SN). SD supports intra-transaction parallelism with less communication overhead than SN, in particular for scan operations. The degree of scan parallelism can dynamically be determined for SD so that selective queries can be processed with minimal communication overhead while large relation scans are spread over many nodes to shorten response time. The nodes that should perform a given operation can also be selected according to the current system state resulting in a high flexibility for dynamic load balancing. This flexibility is particularly valuable for supporting mixed OLTP/query workloads. The communication overhead for parallel query processing is further reduced for SD by the possibility to exchange large intermediate results across shared storage devices rather than over the network. Finally, SD is better positioned than SN to utilize disk arrays and to support parallel query processing for next-generation database applications (object-oriented DBMS) for which partitioning the database among multiple nodes is very difficult.

While many concepts and algorithms for parallel query processing developed for SE and SN can be adapted to the SD environment, substantially more work is needed to fully exploit the SD potential. This is primarily the case for the areas of transaction management, dynamic load balancing and disk allocation. Furthermore, the performance advantages of SD outlined in this paper must be validated by detailed performance studies.

6 References

DG92 DeWitt, D.J., Gray, J.: Parallel Database Systems: The Future of High Performance Database Systems. *Comm. ACM* 35 (6), 85-98, 1992

GDQ92 Ghandeharizadeh, S., DeWitt, D.J., Qureshi, W.: A Performance Analysis of Alternative Multi-Attribute Declustering Strategies. *Proc. ACM SIGMOD Conf.*, 29-38, 1992

GHW90 Gray, J., Horst, B., Walker, M.: Parity Striping of Disc Arrays: Low-Cost Reliable Storage with Acceptable Throughput. *Proc. 16th Int. Conf. on Very Large Data Bases*, 148-161, 1990

Gr91 Gray, J. (ed.): *The Benchmark Handbook for Database and Transaction Processing Systems.* Morgan Kaufmann Publishers, 1991

HR93 Härder, T., Rothermel, K.: Concurrency Control Issues in Nested Transactions. *VLDB Journal* 2 (1), 1993

Li93 Linder, B.: Oracle Parallel RDBMS on Massively Parallel Systems. *Proc. PDIS-93*, 67-68, 1993

MR92 Marek, R., Rahm, E.: Performance Evaluation of Parallel Transaction Processing in Shared Nothing Database Systems, *Proc. 4th Int. PARLE Conf.*, Springer-Verlag, LNCS 605, 295-310, Paris, June 1992

PGK88 Patterson, D.A., Gibson, G., Katz, R.H.: A Case for Redundant Arrays of Inexpensive Disks (RAID). *Proc. ACM SIGMOD Conf.*, 109-116, 1988

Pi90 Pirahesh, H. et al.: Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches. In *Proc. 2nd IEEE Int. Symp. on Databases in Parallel and Distributed Systems*, 1990

Ra91 Rahm, E.: Concurrency and Coherency Control in Database Sharing Systems, Techn. Report 3/91, Univ. Kaiserslautern, Dept. of Comp. Science, Dec. 1991

Ra92 Rahm, E.: A Framework for Workload Allocation in Distributed Transaction Processing Systems. *Journal of Systems and Software* 18, 171-190, 1992

Ra93a Rahm, E.: Empirical Performance Evaluation of Concurrency and Coherency Control for Database Sharing Systems. *ACM Trans. on Database Systems* 18 (2), 333-377, 1993

Ra93b Rahm, E.: Parallel Query Processing in Shared Disk Database Systems for High Performance Database Systems. TR 1/93, Univ. Kaiserslautern, Dept. of Comp. Science (full version of this paper)

RM93 Rahm, E., Marek, R.: Analysis of Dynamic Load Balancing Strategies for Parallel Shared Nothing Database Systems, *Proc. 19th Int. Conf. on Very Large Data Bases*, 1993

Va93a Valduriez, P.: Parallel Database Systems: Open Problems and New Issues. *Distr. and Parallel Databases* 1 (2), 137-165, 1993

Va93b Valduriez, P.: Parallel Database Systems: The Case for Shared-Something. *Proc. 9th Int. Conf. on Data Engineering*, 460-465, 1993

YD91 Yu, P.S., Dan, A.: Comparison on the Impact of Coupling Architectures to the Performance of Transaction Processing Systems. *Proc. 4th Int. Workshop on High Performance Transaction Systems*, 1991