Analysis of Dynamic Load Balancing Strategies for Parallel Shared Nothing Database Systems

Erhard Rahm
Robert Marek

University of Kaiserslautem , Germany
E-mail: { rahm I marek }@informatik.uni-kl.de

Abstract

Parallel database systems have to support both inter-transaction as well as intra-transaction parallelism. Inter-transaction parallelism (multi-user mode) is required to achieve high throughput, in particular for OLTP transactions, and sufficient cost-effectiveness. Intra-transaction parallelism is a prerequisite for reducing the response time of complex and data-intensive transactions (queries). In order to achieve both goals dynamic strategies for load balancing and scheduling are necessary which take the current system state into account for allocating transactions and subqueries to processors and for determining the degree of intra-transaction parallelism. We study the load balancing problem for parallel join processing in Shared Nothing database systems. In these systems, join processing is typically based on a dynamic redistribution of relations to join processors thus making dynamic load balancing strategies feasible. In particular, we study the performance of dynamic load balancing strategies for determining the number of join processors and for selection of the join processors. In contrast to previous studies on parallel join processing, we present a multi-user performance analysis for both homogeneous and heterogeneous/mixed workloads as well as for different database allocations.

1 Introduction

Parallel database systems are the key to high performance transaction and database processing [6]. 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 adhoc queries. To achieve high OLTP throughput, inter-transaction parallelism (multi-user mode) is required in order to overlap transaction deactivations for I/O or remote database requests. Furthermore, single-user mode would result in poor cost-effectiveness since the available processing capacity could not fully be utilized. Intra-transaction (intra-query) parallelism is needed in order to provide short response times for complex queries [23]. OLTP and query performance should scale with the number of nodes: ideally adding processing nodes linearly improves OLTP throughput and query response times.

Unfortunately, supporting both high OLTP throughput and short query response times are partially contradicting subgoals due to increased resource and data contention between the two workload types. Data contention problems may be solved by a multiversion concurrency control scheme which guarantees that read-only queries do not suffer from or cause any lock conflicts [1, 21]. Increased resource contention, on the other hand, is unavoidable since complex queries pose high CPU, memory and disk bandwidth requirements which can result in significant delays for concurrently executing (OLTP) transactions. Furthermore, intra-query parallelism inevitably causes increased communication overhead (compared to a sequential execution on one node) thereby reducing the effective CPU utilization and thus throughput. In addition, it may be difficult to find a physical database allocation supporting both workload types. Efficient OLTP processing can be supported by a clustering of data so that selective queries can be processed with a minimum of communication. Effective parallelization of complex queries, on the other hand, requires a declustering of data across many disks so that many processors can be utilized in parallel to reduce response time.

To limit and control resource contention between concurrent transactions and queries, there is a clear need of dynamic scheduling and load balancing strategies. Within a processing node, local scheduling components have to be extended

**Proceedings of the 19th VLDB Conference,
Dublin, Ireland, 1993**

to control local resource contention, e.g., by adding support for transaction priorities [16, 8, 2]. To effectively utilize a distributed system, the workload must be allocated among the processing nodes such that load balancing is achieved (so that the capacity of different processing nodes is evenly utilized) to limit resource (CPU) contention. At the same time, workload allocation should support a compromise with respect to communication overhead such that both a sufficiently high throughput and intra-transaction parallelism can be achieved. This requires a dynamic query processing approach where the degree of intra-query parallelism as well as the determination of which processing nodes should process a given query are made dependent on the current system state at query run time. As we will show, the optimal degree of intra-transaction parallelism (which yields the best response time) is generally the lower the higher the system is utilized. This is because the communication overhead associated with a high degree of intra-transaction parallelism is less affordable when processors are highly utilized.

In this paper, we study the performance of several static and dynamic load balancing (workload allocation) alternatives for parallel query processing in Shared Nothing systems. Currently, Shared Nothing represents the major architecture for intra-query parallelism and is adopted by several DBMS products and prototypes [6]. Unfortunately, the potential for dynamic load balancing is limited for Shared Nothing because for many operations the execution location is statically determined by the partitioning and allocation of the database among processing nodes. This is particularly the case for scan (selection) operations which are always executed where the data to be processed resides. However, for database operators like join which typically work on derived data (intermediate results), dynamic load balancing becomes feasible by dynamically redistributing the data.

For this reason, our performance (simulation) study primarily concentrates on parallel join processing in multi-user mode. While several previous studies have analysed the performance of parallel join processing (see next section), these studies were all restricted to single-user mode. This corresponds to a best-case situation with little or no resource contention; as a result there is little need for dynamic load balancing in this case (see Section 3). For dynamic load balancing in multi-user mode, we investigate several heuristics for choosing the degree of join parallelism and/or the join processors themselves according to the current system state at query run time. Multi-user experiments will be presented for both homogeneous and heterogeneous (mixed) workloads. We also consider the influence of the database allocation, in particular the degree of declustering (full vs. partial declustering). For comparison purposes, results for static load balancing strategies and single-user mode are also analysed.

The remainder of this paper is organized as follows. The next section contains a brief survey of related studies on load balancing and parallel join processing. In Section 3 we motivate the need for dynamic load balancing strategies by presenting some basic simulation experiments demonstrating that the optimal degree of join parallelism depends on the current system utilization. Section 4 provides an overview of our simulation system. In Section 5 we describe and analyse simulation experiments that were conducted to study the performance of different load balancing strategies for different database and workload configurations. Finally, we summarize the major findings of this investigation.

2 Related Work

A substantial amount of research has been conducted on load balancing in general distributed systems and in distributed operating systems [4, 34, 29]. However, these studies usually assumed that each job can be equally processed by any node and that each job only requires CPU and memory resources. Load balancing is much more complex for distributed database processing since the performance is influenced by additional factors like disk I/O, data contention and communication frequency. For non-parallel (distributed) database processing, a so-called affinity-based workload allocation is generally advisable [33, 24]. It assigns transactions with an affinity to the same database portions to the same processing nodes to support locality of reference and to reduce the communication requirements. Such a workload allocation is primarily concerned with assigning entire transaction requests to processing nodes; a survey of such transaction routing strategies can be found in [24]. For distributed and parallel database processing, an additional load distribution for smaller work granules (subqueries) has to be performed by the nodes' DBMS. As already mentioned, for Shared Nothing this load distribution is largely influenced by the physical database allocation, but parallel processing of some complex query types, in particular join queries, permits a dynamic load balancing.

A number of studies has already addressed load balancing issues for parallel query processing. However, dynamic load balancing was mainly considered for parallel Shared Memory (multiprocessor) DBMS so far [14, 15, 13, 18]. In this case, dynamic load balancing is easily achieved since the operating system can automatically assign the next ready process/subquery to the next free CPU. Furthermore, the shared memory supports very efficient interprocess communication so that the overhead for starting/terminating subqueries is much lower than for Shared Nothing. On the other hand, the number of processors is typically small for Shared Memory (≤ 30) thus restricting the degree of inter-/intra-transaction parallelism and the potential for dynamic load balancing.

For Shared Nothing, physical database design aims at supporting a static form of load balancing for complex queries by declustering relations across many nodes to support a high degree of intra-query parallelism [3, 10]. Such an approach is not only static but also limited to intra-query load balancing. In multi-user mode, the chosen database allocation can easily lead to poor load balancing since the actual workload mix may constantly change while physical database design must be based on an expected average load profile. Another form of static load balancing has been considered in [5] in order to find a processor allocation for inter-operator parallelism (processing of multi-way joins). The processor allocation was already determined at query compile time assuming single-user mode; thus only intra-query load balancing can be achieved.

Dynamic forms of load balancing have been proposed for join processing in order to deal with data skew [31, 30, 7]. These approaches dynamically determine the size of intermediate results in order to redistribute the data among join processors such that they have to perform about the same join work (in order to minimize execution skew). However, this also can only guarantee intra-query load balancing which may easily be destroyed in the case of multi-user mode. Other performance studies of parallel join processing for Shared Nothing (without data skew) also assumed single-user mode, e.g., [26, 27, 22]. The only multi-user performance studies of intra-transaction parallelism for Shared Nothing we are aware of are [9, 19]. However, these papers only considered scan (selection) queries and did not address dynamic load balancing.

3 The Need for Dynamic Load Balancing for Parallel Join Processing

In this section, we present some basic simulation results on parallel join processing to illustrate the need for dynamic load balancing. The results were obtained with a detailed simulator of Shared-Nothing systems to be described in Section 4. Join processing is based on a dynamic redistribution of the relations to be joined. Typically the input data for the join is obtained by scan operations that redistribute their output to a specified number of join processors by applying a hash function on the join attribute. By using the same hash function for the two relations to be joined, it is guaranteed that all matching tuples arrive at the same join processor [26].

Apparently, the number of join processors (degree of join parallelism) is a critical parameter of this approach since it determines the maximal response time speedup compared to a sequential join processing. To study which degree of join parallelism minimizes response time we conducted a number of simulation runs for both single-user and multi-user environments. For this experiment we assumed a join query similar to the Wisconsin *joinABprime* query [12], but with additional selections on both input relations. One relation (A) contains 1 million tuples, the other (Bprime) 100.000 tuples; the join result has the same size as the scan output on the smaller relation. The scans on both relations are supported by a clustered index. The system was assumed to consist of 80 processing nodes; both relations are declustered across 40 disjoint nodes (disks).

Fig. 1 shows the average single-user response times for this join query and system configuration for different degrees of join parallelism (1-80) and scan selectivities. The join processors are selected at random. For each selectivity we have printed the best response time in boldface in Fig. 1 to indicate the optimal number of join processors. One observes that a high number of join processors is most effective for "large" joins, i.e., for high scan selectivity (10%). In this case, response times could continuously be improved by increasing the degree of join parallelism. For small joins (selectivity 0.1%) response times improved only for up to 10 join processors. This is because the work per join processor decreases with the degree of join parallelism, while the communication overhead for redistributing the data increases. Note that the response time improvements are constrained not only by communication delays, but also by the fact that the scan portion of the response times is not improved when increasing the number of join processors.

# of join processors vs. scan selectivity	1	10	20	40	80
10 %	5461	1293	897	786	**741**
1 %	725	239	**215**	219	255
0.1 %	182	**140**	144	162	201

Figure 1: Single-user response time (in ms) for different degrees of join parallelism and scan selectivities

We observed that in single-user mode when the entire system is at the disposal of a single query, the optimal degree of join parallelism can statically be determined at query compile time (if no data skew occurs). This is because the optimal number of join processors is mainly determined by the ratio of communication overhead and useful work per node and thus by rather static parameters such as the cost of message passing, CPU speed, network capacity, database allocation, relation sizes and scan selectivity. Provided these basic parameters are known or can be determined experimentally, we can thus use an analytical formula to calculate the approximate response time for a given number of join processors. This also allows calculation of the optimal degree of join parallelism by setting the derivative of the response time formula to zero, similarly as described in [32].

For the multi-user experiment, we varied the arrival rate for our join query. The resulting response time results for different degrees of join parallelism and 0.1% scan selectivity are shown in Fig. 2. The results show that multi-user mode significantly increases query response times due to increased resource (CPU) contention and higher communication overhead. Furthermore, the effectiveness of join parallelism increasingly deteriorates with growing arrival rates (queries per second, QPS). As a result, the optimal degree join parallelism for single-user mode does not yield the best response times in multi-user mode. Rather the optimal degree of join parallelism depends on the arrival rate and thus on the current system utilization; it becomes the lower the higher the system is utilized. This is because the communication overhead increases with the number of join processors which is the less affordable the more restricted the CPU resources are.

For the join with 0.1% scan selectivity (Fig. 2) the optimal join parallelism was only 1 (sequential execution) for an arrival rate of 55 QPS. For this arrival rate, the single-user optimum of 10 join processors results in a response time that is 2.7 times higher than for the multi-user optimum.

# of join processors vs. query arrival rate	1	2	6	8	10	20
single-user mode	182	162	147	141	**140**	144
15 QPS	204	184	**179**	192	198	257
35 QPS	249	**240**	261	338	394	894
55 QPS	**310**	325	381	604	856	----

Figure 2: Multi-user response time (in ms) for different degrees of join parallelism and arrival rates (selectivity 0.1 %)

Our experiment shows that the degree of join parallelism may be statically determined for single-user mode, but that there is a strong need for dynamic load balancing for parallel join processing in multi-user mode. This leads to the problem of how the degree of join parallelism can be determined dynamically? For this purpose, we have implemented a simple heuristic in our Shared Nothing simulator. It uses the optimal single-user join parallelism as the default which is then dynamically decremented according to the system (CPU) utilization at query run time. Apart from dynamically determining the degree of join parallelism p, we are also studying several alternatives for selecting the p join processors from the available processing nodes. In Section 5, the various load balancing strategies are described in more detail when we present the simulation results.

4 Simulation Model

Our simulation system models the hardware and database processing logic of a generic Shared Nothing DBMS architecture. The system has been implemented using the discrete event simulation language DeNet [17]. Our system consists of three main components: *workload generation, workload allocation* and *processing subsystem* (Fig. 3). The workload generation component models user terminals and generates work requests (transactions, queries). The workload allocation component assigns these requests to the processing nodes (processing elements, PE) where the actual transaction/query processing takes place. We first describe workload generation and allocation; in 4.2 we sketch the modelling of workload processing.

4.1 Workload Generation and Allocation

Database model

Our database model supports four object granularities: database, partitions, pages and objects (tuples). The database is modeled as a set of partitions. A partition may be used to represent a relation, a relation fragment or an index structure. It consists of a number of database pages which in turn consist of a specific number of objects (tuples, index entries). The number of objects per page is determined by a blocking factor which can be specified on a per-partition basis. Differentiating between objects and pages is important in order to study the effect of clustering which aims at reducing the number of page accesses (disk I/Os) by storing related objects into the same page. Furthermore, concurrency control may now be performed on the page or object level. Each relation can have associated clustered or unclustered B^+-tree indices.

We employ a horizontal data distribution of partitions (relations and indices) at the object level controlled by a relative distribution table. This table defines for every partition P_j and processing element PE_i which portion of P_j is allocated to PE_i.This approach models range partitioning and supports full declustering as well as partial declustering.

Workload generation

Our simulation system supports heterogeneous workloads consisting of several query and transaction types. Queries correspond to transactions with a single database operation (e.g., SQL statement). Currently we support the following query types: relation scan, clustered index scan, non-clustered index scan, two-way join queries, multi-way join queries, and update statements (both with and without index support). We also support the debit-credit benchmark workload (TPC-B) and the use of real-life database traces [19]. The simulation system is an open queuing model and allows definition of an individual arrival rate for each transaction and query type.

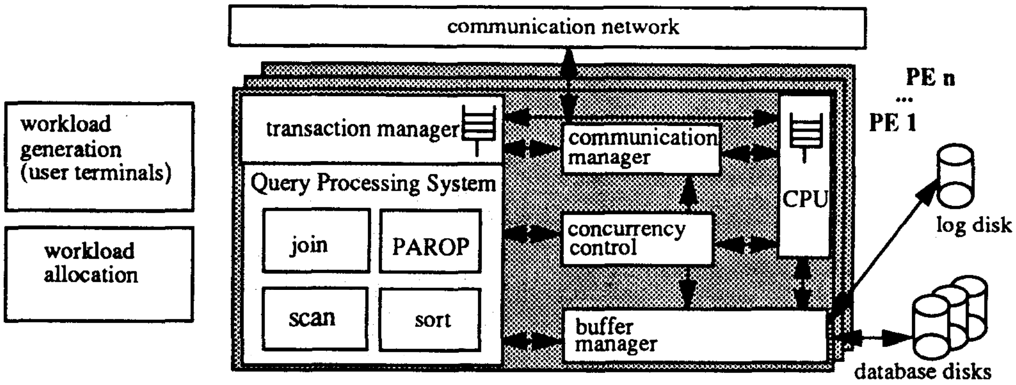


Figure 3: Gross structure of the simulation system

For parallel join processing we have implemented a representative strategy based on hash partitioning. It applies a hash function on the join attribute to partition both input relations (scan output relations) to a specific number of join processors (dynamic data redistribution). This hash partitioning guarantees that tuples with the same join attribute value are assigned to the same join processor. This approach has the advantage that it offers a high potential for dynamic load balancing since the number and selection of join processors constitute dynamically adjustable parameters. We also support the special cases where one or both relations are partitioned on the join attribute so that only one or no relation may have to be redistributed (see Section 5). This reduces the communication overhead for join processing but may limit the potential for dynamic load balancing. For local join processing we have modelled a sort-merge algorithm. At each join processor the input relations are first sorted on the join attribute. The sorted relations are then scanned and matching tuples are added to the output stream. The complete join result is obtained by merging the results of the distributed local joins.

In the query graphs of our model, parallelism is expressed by means of a so-called *parallelization* meta-operator (PAROP). This operator implements inter- as well as intra-operator parallelism and encapsulates all parallelism issues, similar to the exchange operator used in the Volcano prototype [11]. In particular, the PAROP operator comprises two basic parallelization functions: a *merge* function which combines several parallel data streams into a single sequential stream, and a *split* function which is used to partition or replicate the stream of tuples produced by a relational operator [6].

Workload allocation

Two forms of workload allocation have to be distinguished. First, each incoming transaction (query) is assigned to one PE (acting as the coordinator for the transaction) according to a placement strategy. Our simulation system supports different placement strategies, in particular a random allocation or the use of a routing table[1]. The second form of workload allocation deals with the assignment of suboperations to processors during query processing and depends on the operators to be executed. For scan operators, the processor allocation is always based on a relation's data allocation. For join processing, we support several static and dynamic strategies for determining the degree of join parallelism and for allocating the join processes to processors (e.g., random allocation or based on the current CPU utilization). More details are provided in Section 5.

4.2 Workload Processing

The processing component models the execution of a workload on a Shared Nothing system with an arbitrary number of PE connected by a communication network. Each PE has access to private database and log files allocated on external storage devices (disks). Internally, each PE is represented by a transaction manager, a query processing system, a buffer manager, a concurrency control component, a communication manager and a CPU server (Fig. 3).

The transaction manager controls the (distributed) execution of transactions. The maximal number of concurrent transactions (inter-transaction parallelism) per PE is controlled by a multiprogramming level. Newly arriving transactions must wait in an input queue until they can be served when this maximal degree of inter-transaction parallelism is already reached. The query processing system models basic relational operators (sort, scan, join) as well as the PAROP meta-operator (see above).

Execution of a transaction starts with the BOT processing (begin of transaction) entailing the transaction initialization overhead. For each database operation of the transaction, the actual query processing is performed according to the relational query tree. Basically, the relational operators process local input streams (relation fragments, intermediate results)

[1] The routing table specifies for every transaction type T_j and processing element PE_i which percentage of transactions of type T_j will be assigned to PE_i. It can be used to achieve an affinity-based transaction routing.

and produce output streams. The PAROP operators indicate when parallel sub-transactions have to be started and perform merge and split functions on their input data streams. An EOT step (end of transaction) triggers two-phase commit processing involving all PE that have participated during execution of the respective transaction. We support the optimization proposed in [20] where read-only sub-transactions only participate in the first commit phase.

CPU requests are served by a single CPU per PE. The average number of instructions per request can be defined separately for every request type. To accurately model the cost of query processing, CPU service is requested for all major steps, in particular for transaction initialization (BOT), for object accesses in main memory (e.g., to compare attribute values, to sort temporary relations or to merge multiple input streams), I/O overhead, communication overhead, and commit processing.

For concurrency control, we employ distributed strict two-phase locking (long read and write locks). The local concurrency control manager in each PE controls all locks on the local partition. Locks may be requested either at the page or object level. A central deadlock detection scheme is used to detect global deadlocks and initiate transaction aborts to break cycles.

Database partitions can be kept memory-resident (to simulate main memory databases) or they can be allocated to a number of disks. Disks and disk controllers have explicitly been modelled as servers to capture I/O bottlenecks. Disks are accessed by the buffer manager component of the associated PE. The database buffer in main memory is managed according to a global LRU (Least Recently Used) replacement strategy.

The communication network provides transmission of message packets of fixed size. Messages exceeding the packet size (e.g., large sets of result tuples) are disassembled into the required number of packets.

5 Simulation Experiments and Results

Our experiments concentrate on the performance of parallel join processing in multi-user mode. For comparison purposes, single-user experiments have also been conducted. The focus of the study is to compare different static and dynamic load balancing alternatives for determining the degree of join parallelism and for selection of the join processors. For this analysis, we consider different database allocations with full and partial declustering and the use of dedicated join processors with no associated permanent data. These separate join processors may be able to improve load balancing since they have no scan operations to execute. Two load profiles are studied for multi-user mode: a homogeneous workload only consisting of join queries that are concurrently executed as well as a heterogeneous (mixed) workload with both short OLTP transactions and join queries.

In the next subsection, we provide an overview of the parameter settings that are used for these experiments. In 5.2, we describe the single-user experiments. Multi-user experiments for the homogeneous and heterogeneous workload are analyzed in 5.3 and 5.4, respectively.

5.1 Workload Profile and Simulation Parameter Settings

Fig. 4 shows the major database, query and configuration parameters with their settings. Most parameters are self-explanatory, some will be discussed when presenting the simulation results. The join queries used in our experiments perform two scans (selections) on the input relations A and B in parallel and join the corresponding results. The *A* relation contains 1 million tuples, the *B* relation 250.000 tuples. The selections on *A* and *B* reduce the size of the input relations according to the selection predicate's selectivity (percentage of input tuples matching the predicate). Both selections employ clustered indices. The join result has the same size as the scan output on B. Scan selectivity on both relations is set to 0.25%. The number of processing nodes is varied between 10 and 80.

We investigate three different strategies for database partitioning and allocation:

- *Full Declustering (FD):*
  Both relations are uniformly declustered across all PE.
- *Partial Declustering (PD):*
  Both relations are uniformly declustered across disjoint sets of PE. To support a static load balancing for scan operations, each PE is assigned the same number of tuples. As a result the larger relation A is declustered across 80% of the PE, while the remaining 20% of the PE hold tuples of relation B.
- *Separate Join Processors (SJP):*
  In this case we reserve 20 processors for join processing and use partial declustering for allocating the two relations across the remaining PE (i.e., A and B reside on disjoint PE). This allocation is only studied for configurations with 20 and more PE; in the case of 20 PE only 10 processors are reserved for join processing.

The number of dedicated join processors in the SJP allocation was set to 20 since this was determined to be the optimal degree of join parallelism for our join query in single-user mode when both relations have to be redistributed.

Parameters for the I/O (disk) subsystem have been chosen so that no bottlenecks occurred (sufficiently high number of disks and controllers). The duration of an I/O operation is composed of the controller service time, disk access time and transmission time. The parameter settings for the communication network have been chosen according to the EDS prototype [28].

5.2 Single-User Experiments

In single-user mode we employed only static strategies for allocating the join work. Fig. 5a shows the average response times for our join query in the case of full declustering, Fig. 5b for partial declustering and the use of separate join processors. Parallel join processing is either performed on the optimal number of join processors (20 for #PE ≥ 20, 10 otherwise) or on all PE holding tuples of relation A. Except for SJP, join processors are selected at random when not all PE are used for join processing; for SJP join processing is performed on the dedicated join processors. In most cases, the scan output of both relations was completely redistributed and sent to the join processors. We also considered two special cases permitting a smaller communication overhead for data redistribution. For full declustering, we additionally studied the case when no redistribution is necessary because both relations are partitioned on the join attribute and assigned to the same set of PE. For partial declustering, we included results for the case when only the smaller relation B needs to be redistributed, assuming relation A is already partitioned on the join attribute and the join is performed on the A nodes.

Fig. 5 shows that the use of intra-query parallelism for scan and join processing reduces response times for up to 40 PE (20 PE in the case of full declustering when both relations are redistributed). However, no linear speedup is achieved since the communication overhead for starting/terminating suboperations and data redistribution is comparatively high due to the high selectivity; for more than 40 PE the increasing communication overhead prevents further response time improvements. For full declustering and single-user mode, performance is primarily determined by the communication overhead and not by the potential for dynamic load balancing. Thus the best response times were achieved for the special case where no data redistribution was necessary for join processing. In the case when both relations are redistributed, choosing the optimal number of join processors (20) outperforms the case where the join is performed on all nodes holding fragments of relation A. This is because the latter strategy causes more communication overhead for data redistribution for more than 20 PE since the join is then performed on more than 20 PE (80% of all PE). Similar observations hold for partial declustering. However the special case where only relation B is redistributed performs best

Configuration	settings	Database/Queries	settings
number of PE (#PE)	10, 20, 40, 60, 80	**relation A:**	(200MB)
CPU speed per PE	20 MIPS	#tuples	1.000.000
		tuple size	200 bytes
avg. no. of instructions:		blocking factor	40
BOT	25000	index type	clustered B^+-tree
EOT	25000	storage allocation	disk
I/O	3000	allocation to PE	FD, PD, SJP
send message	5000		
receive message	10000	**relation B:**	(50MB)
copy 8KB message	5000	#tuples	250.000
scan object reference	1000	tuple size	200 bytes
join object reference	500	blocking factor	40
sort n tuples	$n \log_2(n) * 10$	index type	clustered B^+-tree
		storage allocation	disk
buffer manager:		allocation to PE	FD, PD, SJP
page size	8 KB		
buffer size per PE	250 pages (2 MB)	**intermediate results:**	
		storage allocation	disk
disk devices:		**join queries:**	
controller service time	1 ms (per page)	access method	via clustered index
transmission time per page	0.4 ms	input relations sorted	FALSE
avg. disk access time	15 ms	scan selectivity	0.25%
		no. of result tuples	625
communication network:		size of result tuples	400 bytes
		arrival rate	single-user, multi-user (varied)
packet size	128 bytes	query placement	random (uniformly over all PE)
avg. transmission time	8 microsec	join parallelism	static / dynamic (DJP)
		selection of join processors	random / dynamic (LUP, ALUP)

Figure 4: System configuration, database and query profile.

only for up to 40 PE; for larger configurations it is outperformed by the strategy redistributing both relations but limiting join processing to 20 PE. This was because the high number of join processors in the former strategy causes a comparatively high number of messages for redistributing relation B in addition to the high communication overhead for startup and termination of join processing.

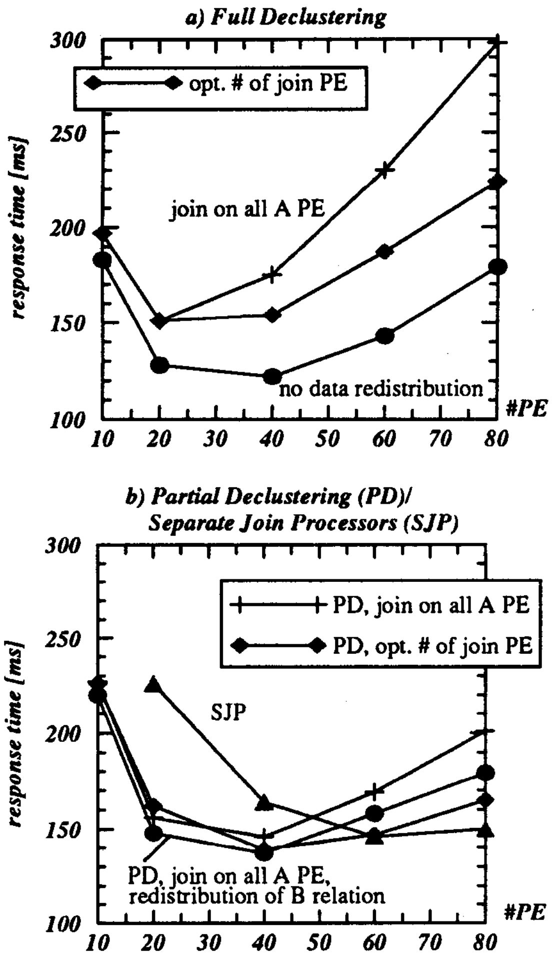


Figure 5: Single-user results

The use of separate join processors (SJP) did not prove useful since all PE were lightly loaded in single-user mode so that they are all good candidates for join processing. However, reserving 20 PE for join processing results in a smaller degree of scan parallelism since the two relations had to be assigned to fewer nodes. Hence, SJP response times were substantially worse than for FD or PD and the SJP optimum lies at 60 PE rather than 40 PE. Full declustering achieved better response times than partial declustering for a lower number of nodes, while PD outperforms FD for more than 20 PE. This is because FD allows a higher degree of scan parallelism, but also leads to a higher communication overhead for starting the scan operations and redistributing the scan output. For a higher number of nodes the reduced communication overhead of PD is more significant than the lower scan parallelism. This is also due to the comparatively low number of tuples to be processed per scan node for a higher number of PE.

5.3 Multi-user experiments with homogeneous workload

The homogeneous workload still consists of a single (join) query type, but we employ intra-query parallelism in combination with inter-query parallelism. Since we want to support not only short response times but also good throughput we increase the query arrival rate proportionally with the number of PE. We first present multi-user results for some of the static workload allocation strategies used in the preceding section. Afterwards we analyze the effectiveness of four dynamic load balancing strategies.

Static load balancing experiments

Fig. 6 compares the single-user with multi-user response times for arrival rates of 0.4 and 0.5 QPS per PE in the case of full declustering for both relations. With respect to join processing, results for the special case with no data redistribution are shown as well as for a redistribution of both relations. In the latter case, we always use the optimal single-user join parallelism (20 for #PE ≥ 20) and randomly select the join processors. One observes that for the considered arrival rates, the multi-user results are not much higher than for single-user mode if the joins can locally be performed without any data redistribution. While the communication overhead for redistributing both relations only causes a modest response time increase in single-user mode, response times rapidly deteriorate in multi-user mode for more than 20 PE. This is mainly caused by three factors. First, the use of full

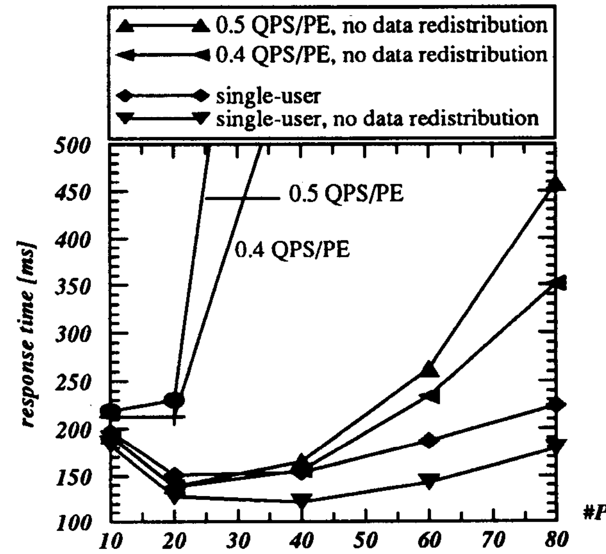


Figure 6: Multi-user results for full declustering (static load balancing)

declustering causes a maximal communication overhead for scan and data redistribution as discussed above. Second, since we increase the total query load proportionally with the number of PE the communication overhead even increases quadratically with more processors. Thus, above a certain number of PE excessive resource contention is introduced. Finally, load balancing is static and does not consider the current system utilization, e.g., for determining the degree of join parallelism.

To analyse the impact of the database allocation in multi-user mode, we compare the full declustering results with partial declustering and the use of separate join processors (Fig. 7). For this purpose, we only consider the general case with redistribution of both relations for an arrival rate of 0.5 QPS per PE. Fig. 7 shows that partial declustering clearly outperforms full declustering for more than 10 PE due to its lower communication overhead which is much more significant in multi-user than in single-user mode. The use of separate join processors is slightly more effective than in single-user mode, but is still outperformed by PD and FD. The smaller number of scan processors for SJP allows for a reduced communication overhead, but this cannot fully compensate the smaller degree of scan parallelism. SJP also suffers from load imbalances between the scan and join processors, in particular for more than 40 PE when the join processors become overloaded[2].

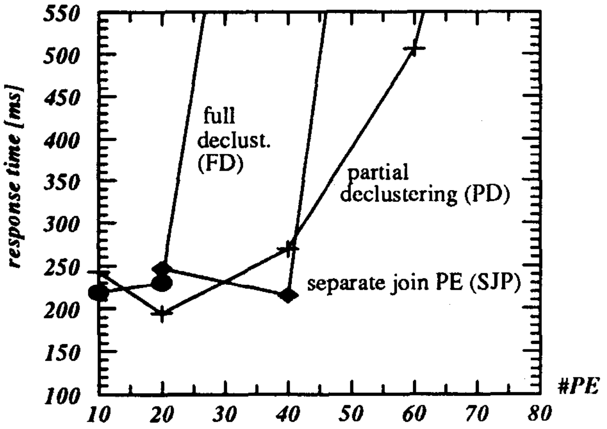


Figure 7: Multi-user results for different database allocations (0.5 QPS/PE)

Dynamic load balancing experiments

The preceding multi-user experiments showed that static load balancing leads to poor performance for a higher number of PE when both relations are to be redistributed. We now study whether performance can be improved by dynamic load balancing. The primary metric we use for dynamically adapting the degree of join parallelism and for selecting the join processors is the current CPU utilization of the processors. For this purpose we assume that a designated control node is periodically informed by the PE about their current utilization. During the execution of a query, information on the current CPU utilization is requested from the control node in order to support a dynamic load balancing. The following four dynamic strategies have been implemented for parallel join processing:

- *Dynamic adaptation of the degree of join parallelism (DJP)*
  This strategy only determines the number of join processors dynamically; selection of the join processors from the available PE is at random. We use the single-user optimum $p_{su\text{-}opt}$ as the maximal degree of parallelism for multi-user mode and reduce this value according to the current system utilization. We tested several alternatives for finding a good multi-user degree of join parallelism p_{mu} and finally used the following formula:
  $$P_{mu} = P_{su\text{-}opt}\,(1 - u^3).$$
  In this formula, u denotes the current average CPU utilization of all PE obtained from the control node. For an average CPU utilization of 50% ($u = 0.5$), this approach reduces the single-user value by 12.5%; for $u = 0.9$ the degree of join parallelism is reduced by about a factor 4. The formula reflects our observation that for a low CPU utilization ($u < 0.5$), reducing the degree of join parallelism is more detrimental to performance than the communication overhead associated with the optimal single-user degree of join parallelism. For $u > 0.5$, on the other hand, communication overhead must be reduced to keep resource contention acceptable.
- *Join processing on least utilized processors (LUP)*
  In this approach, the degree of join parallelism is statically determined (e.g., $P_{su\text{-}opt}$) but the join processors are selected dynamically. We simply select the least utilized processors as join processors.
- *Adaptive LUP (ALUP)*
  This strategy is an adaptive variation of the previous one which exhibited an undesirable behavior. We observed that the simple LUP policy tends to select the same join processors for two consecutive queries (causing load imbalances) since information on CPU utilization is updated only periodically. The ALUP strategy tries to correct the problem by artificially increasing the utilization of those processors at the control node which have been selected for join processing. This makes it less likely that following queries choose the same join processors.
- *Combined dynamic strategy (DJP + ALUP)*
  This combined strategy dynamically determines the degree of join parallelism p_{mu} according to the DJP policy. In addition the p_{mu} join processors are chosen according to the ALUP approach.

Simulation results for these strategies are shown in Fig. 8 for an arrival rate of 0.5 QPS per PE and partial declustering. For comparison, we have also included the result for static load

[2] CPU requirements for the join portion of our query are slightly higher than for the scan portion. For more than 40 PE however, there are more scan than join processors for SJP.

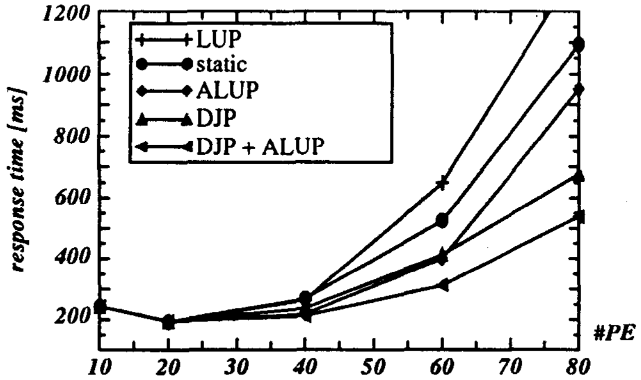


Figure 8: Multi-user results with dynamic load balancing for partial declustering (0.5 QPS per PE)

balancing (random selection of $p_{su\text{-}opt}$ join processors). Fig. 8 shows that the dynamic strategies clearly outperform static load balancing for more than 40 PE, except the simple join processor selection policy LUP. This was due to the above-mentioned problem of LUP which prevented a more effective load balancing than for random allocation. Note, that even the static case allowed for a comparatively good load balancing for the homogeneous workload. This is because the scan workload is evenly balanced for the chosen database allocations and random selection of join processors also achieves a balanced *average* CPU utilization. However, the *actual* CPU utilization may still vary significantly for different PE and this fact is utilized by the ALUP policy. This adaptive strategy could substantially improve response times for higher utilization levels (large number of PE) by selecting lowly utilized processors for join processing to reduce resource contention. The DJP was even more effective since it reduced the communication overhead by selecting fewer join processors for a higher number of PE. The combined dynamic policy was clearly the best load balancing strategy. It could actually combine the advantages of the DJP and ALUP policies so that communication overhead and resource contention are reduced. The fact that response times kept comparatively low despite the fact that arrival rates increase proportionally with the number of PE shows that the combined dynamic strategy was able to support both a linear throughput increase as well as short response times.

In Fig. 9 we compare the effectiveness of the combined dynamic load balancing strategy for full declustering, partial declustering and the use of separate join processors. A comparison with Fig. 7 shows that in all three cases the dynamic strategy substantially improves response times compared to static load balancing. This was particularly the case for the use of separate join processors which were overloaded for more than 40 PE under static load balancing. The dynamic strategy eliminated the join bottleneck by also considering the scan processors for join processing so that an even CPU utilization could be achieved across all PE. For more than 40 PE, SJP outperforms partial declustering since it incurs a smaller communication overhead for scan processing. Again, full declustering incurs the highest communication overhead thus causing a high resource contention for a larger number of PE even under optimal load balancing.

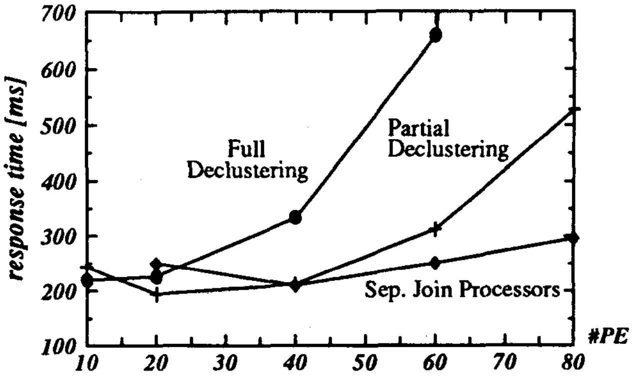


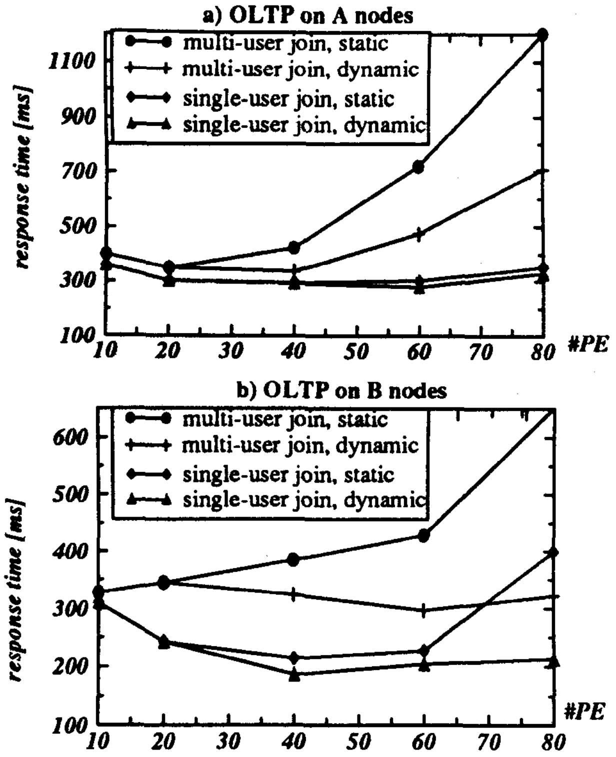
Figure 9: Dynamic load balancing (DJP + ALUP) for different database allocations (0.5 QPS/PE)

5.4 Multi-user experiments with heterogeneous workloads

In the homogeneous multi-user experiments, a comparatively good load balancing was already supported by the chosen database allocation. Furthermore, the use of a single query type resulted in a similar load situation at the different processing nodes (except for SJP). We now study the effectiveness of dynamic load balancing for heterogeneous workloads consisting of one OLTP transaction type and our join query. For OLTP processing, we assume a simple transaction type accessing only one relation (A or B) and that an affinity-based routing can achieve a largely local processing (similar to debit-credit). For the concurrent execution of join queries, we study single-user join processing (only one join query is executed at a time concurrently with OLTP) and multi-user join processing.

Fig. 10 shows the average join response times for two mixed workloads differing in whether the OLTP transaction type is accessing relation A (Fig. 10a) or relation B (Fig. 10b). In both cases we assume a partial declustering of the relations and an OLTP transaction rate of 150 TPS (transactions per second) per A (B) node. The OLTP workload causes a CPU utilization of about 50% per A (B) node. For multi-user join processing, we use an arrival rate of 0.1 QPS per PE. Static load balancing for the join query refers to the case where the join is performed on $p_{su\text{-}opt}$ processors that are randomly selected. For dynamic load balancing we use the combined strategy which dynamically adapts the degree of join parallelism (DJP) and which selects the join processors based on the current CPU utilization (ALUP).

We observe that for the mixed workloads dynamic load balancing is in deed even more effective than for the homogeneous load, in particular for multi-user join processing. Again, the differences between static and dynamic load bal-


a) OLTP on A nodes

b) OLTP on B nodes

Figure 10: Static vs. dynamic load balancing (DJP + ALUP) for mixed workloads

ancing increase with the number of PE. This is because the communication overhead per join query increases with more processors and thus the average CPU utilization. The absolute join response times are substantially higher for OLTP processing on the A nodes (Fig. 10a) since we have the fourfold OLTP throughput in this case and thus a reduced potential for load balancing. For OLTP processing on the B nodes, the A nodes are only lightly loaded and therefore ideally suited for join processing. This could be utilized by our dynamic load balancing strategy and caused a substantial response time improvement for more than 20 PE (Fig. 10b). For 80 PE, dynamic load balancing could cut response times by half (100% improvement) compared to static load balancing.

6 Summary

We have presented a simulation study of parallel join processing in Shared Nothing database systems. In contrast to previous studies, we focussed on the performance behavior in multi-user mode since we believe this will be the operating mode where parallel query processing must be successful in practice. Multi-user mode means that only limited resources are available for query processing and that both response time and throughput requirements must be met. This necessitates dynamic scheduling and load balancing strategies for assigning relational operators during query processing.

In contrast to scan operations, parallel join strategies offer a high potential for dynamic load balancing. This is because joins are generally performed on intermediate results which are dynamically redistributed among several join processors to perform the join in parallel. The number of join processors (degree of join parallelism) and the selection of these processors represent dynamically adjustable parameters. Our experiments demonstrated that effectively parallelizing join operations is much simpler in single-user than in multi-user mode. In single-user mode the optimal degree of join parallelism is largely determined by static parameters known at query compile time, in particular the database allocation, relation sizes and scan selectivity. Selection of the join operators is also easy since all processors are lowly utilized in single-user-mode.

In multi-user mode, the optimal degree of join parallelism depends on the current system state and is the lower the higher the nodes are utilized. Using static load balancing strategies is therefore not appropriate for join processing in multi-user mode and was shown to deliver sub-optimal performance. We therefore studied four simple dynamic load balancing strategies for dynamically determining the degree of join parallelism and for selection of the join processors. Most effective was a combined strategy which adjusts both parameters according to the current load situation. It determines the multi-user join parallelism by reducing the optimal single-user join parallelism according to the current CPU utilization. Join processing is assigned to the least utilized processors. To avoid that consecutive queries select the same processors for join processing, we found it necessary to artificially increase the utilization of newly selected join processors to account for the delayed updating of information on the current CPU utilization. With the dynamic strategy it was possible to keep join response times low while increasing throughput linearly with the number of nodes. The effectiveness of the dynamic load balancing strategy was particularly pronounced for mixed workloads consisting of short OLTP transactions and complex join queries.

While the dynamic adaptation of the degree of join parallelism was able to reduce the communication overhead, the communication requirements and thus the potential for load balancing are largely influenced by the static database allocation. We studied different configurations with a full and partial declustering of relations and the use of separate join processors. In multi-user mode, full declustering of relations is generally not acceptable for higher number of nodes. This is because the increased potential for scan parallelism is then outweighed by the high communication overhead which is the less affordable the higher the system is utilized. The use of separate join processors can improve the potential for dynamic load balancing since these processors have no scan work to perform. However, as our results for mixed workloads have shown a similar potential for dynamic load balancing may also be achieved without separate join processors since the current utilization of the nodes may substantially differ.

In future work, we will study further aspects of parallel query processing in multi-user mode that could not be covered in this paper. In particular, we plan to investigate the impact of data skew in multi-user mode. Furthermore, we will study dynamic load balancing strategies for parallel Shared Disk systems. These systems are not based on a static database allocation among nodes so that there is a high load balancing potential even for scan processing [25].

7 References

[1] Bober, P.M., Carey, M.J.: On Mixing Queries and Transactions via Multiversion Locking. *Proc. 8th IEEE Data Engineering Conf.*, 535-545, 1992

[2] Brown, K.P. et al.: Resource Allocation and Scheduling for Mixed Database Workloads. TR 1095, Univ. of Wisconsin, Madison, 1992

[3] Copeland, G., Alexander, W., Boughter, E., Keller, T.: Data Placement in Bubba. *Proc. ACM SIGMOD Conf.*, 99-108, 1988

[4] Casavant, T.L., Kuhl, J.G.: A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems, *IEEE Trans. Software Eng.* 14, 141-154, 1988

[5] Chen, M., Yu, P.S., Wu, K.: Scheduling and Processor Allocation for Parallel Execution of Multi-Join Queries. *Proc. 8th IEEE Data Engineering Conf.*, 58-67, 1992

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

[7] DeWitt, D.J., Naughton, J.F., Schneider, D.A., Seshadri, S.: Practical Skew Handling in Parallel Joins. *Proc. of the 18th Int. Conf. on Very Large Data Bases*, 1992

[8] Englert, S.: Load Balancing Batch and Interactive Queries in a Highly Parallel Environment. *Proc. IEEE Spring CompCon Conf.*, 110-112, 1991

[9] Ghandeharizadeh, S., DeWitt, D.J.: A Multiuser Performance Analysis of Alternative Declustering Strategies. *Proc. 6th IEEE Data Engineering Conf.*, 466-475, 1990

[10] Ghandeharizadeh, S.: Physical Database Design in Multiprocessor Systems. Ph.D. Thesis, Univ. of Wisconsin-Madison, 1990

[11] Graefe, G.: Encapsulation of Parallelism in the Volcano Query Processing System. *Proc. ACM SIGMOD Conf.*, 102-111, 1990

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

[13] Hong, W.: Exploiting Inter-Operation Parallelism in XPRS. *Proc. ACM SIGMOD Conf.*, 19-28, 1992

[14] Hong, W., Stonebraker, M.: Optimization of Parallel Query Execution Plans in XPRS. *Proc. 1st Int. Conf. on Parallel and Distributed Information Systems*, 218-225, 1991

[15] Hirano, Y., Satoh, T., Inoue, U., Teranaka, K.: Load Balancing Algorithms for Parallel Database Processing on Shared Memory Multiprocessors. *Proc. 1st Int. Conf. on Parallel and Distributed Information Systems*, 210-217, 1991

[16] Jauhari, R., Carey, M.J., Livny, M.: Priority-Hints: An Algorithm for Priority-Based Buffer Management. *Proc. 16th Int. Conf. on Very Large Data Bases*, 708-721, 1990

[17] Livny, M.: DeNet Users's Guide, Version 1.5. Computer Science Department, University of Wisconsin, Madison, 1989.

[18] Lu, H., Tan, K.: Dynamic and Load-Balanced Task-Oriented Database Query Processing in Parallel Systems. *Proc. EDBT*, LNCS 580, 357-372, 1992

[19] Marek, R., Rahm, E.: Performance Evaluation of Parallel Transaction Processing in Shared Nothing Database Systems, *Proc. 4th Int. PARLE Conf.* (Parallel Architectures and Languages Europe), Springer-Verlag, Lecture Notes in Computer Science 605, 295-310, 1992

[20] Mohan, C., Lindsay, B., Obermarck, R.: Transaction Management in the R* Distributed Database Management System. *ACM Trans. on Database System 11* (4), 378-396, 1986

[21] Mohan, C., Pirahesh, H., Lorie, R.: Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions. *Proc. ACM SIGMOD Conf.*, 124-133, 1992

[22] Murphy, M.; Shan, M.: Execution Plan Balancing. *Proc. 1st Int. Conf. on Parallel and Distributed Information Systems*, 1991

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

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

[25] Rahm, E.: Parallel Query Processing in Shared Disk Database Systems. Techn. Report, Univ. of Kaiserslautern, 1993

[26] Schneider, D.A., DeWitt, D.J.: A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. *Proc. ACM SIGMOD Conf.*, 110-121, 1989.

[27] Schneider, D.A., DeWitt, D.J.: Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines. *Proc. 16th Int. Conf. on Very Large Data Bases*, 469-480, 1990

[28] Skelton, C.J. et al.: EDS: A Parallel Computer System for Advanced Information Processing, *Proc. 4th Int. PARLE Conf.* (Parallel Architectures and Languages Europe), Springer-Verlag, Lecture Notes in Computer Science 605, 3-18, 1992

[29] Shivaratri, N.G., Krueger, P., Singhal, M.: Load Distributing for Locally Distributed Systems. *IEEE Computer*, 33-44, Dec. 1992

[30] Walton, C.B; Dale A.G.; Jenevein, R.M.: A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. *Proc. 17th Int. Conf. on Very Large Data Bases*, 537-548, 1991.

[31] Wolf, J.L., Dias, D.M., Yu, P.S., Turek, J.: An Effective Algorithm for Parallelizing Hash Joins in the Presence of Data Skew. *Proc. 7th IEEE Data Engineering Conf.*, 200-209, 1991

[32] Wilschut, A.; Flokstra, J.; Apers, P.: Parallelism in a Main-Memory DBMS: The performance of PRISMA/DB. *Proc. 18th Int. Conf. on Very Large Data Bases*, 521-532, 1992

[33] Yu, P.S., Cornell, D.W., Dias, D.M., Iyer, B.R.: Analysis of Affinity-based Routing in Multi-system Data Sharing. *Performance Evaluation* 7 (2), 87-109, 1987

[34] Zhou, S.: A Trace-Driven Simulation Study of Dynamic Load Balancing. *IEEE Trans. Software Eng.* 14, 1327-1341, 1988